O'REILLY®

TURING 图灵程序设计丛书



客户端存储技术

Client-Side Data Storage

- 为服务器减压，提升访问速度
- 让离线应用程序得以真正实现

[美] Raymond Camden 著
马德奎 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

马德奎

数据库开发工程师，主要从事与机械制造行业MES、物料拉动相关的数据库设计及计算逻辑开发。

TURING 图灵程序设计丛书

客户端存储技术

Client-Side Data Storage

[美] Raymond Camden 著
马德奎 译

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（C I P）数据

客户端存储技术 / (美) 雷蒙德·卡姆登
(Raymond Camden) 著 ; 马德奎译. -- 北京 : 人民邮电
出版社, 2017.3
(图灵程序设计丛书)
ISBN 978-7-115-45014-2

Ⅰ. ①客… Ⅱ. ①雷… ②马… Ⅲ. ①存储技术
Ⅳ. ①TP333

中国版本图书馆CIP数据核字(2017)第036578号

内 容 提 要

客户端数据存储赋予浏览器快速访问数据的能力，从而节省网络流量并减轻服务器的压力，同时使离线应用程序得以真正实现。本书从实用角度出发，以丰富的示例代码介绍 Cookie、Web 存储、IndexedDB 等多种客户端存储技术，以及用于简化客户端存储的 JavaScript 库，如 Lockr、Dexie 和 localForage 等。

本书适合所有 Web 开发人员阅读。

◆ 著　[美] Raymond Camden
译　马德奎
责任编辑　朱　巍
执行编辑　谢婷婷　吴威娜
责任印制　彭志环

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：6.5
字数：160千字　2017年 3 月第 1 版
印数：1 – 3 500册　2017年 3 月北京第 1 次印刷

著作权合同登记号　图字：01-2016-9376号

定价：39.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

© 2016 by Raymond Camden.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2017. Authorized translation of the English edition, 2017 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2016。

简体中文版由人民邮电出版社出版，2017。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

前言

排版约定

本书使用了下列排版约定。

- 楷体

  表示新术语。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 等宽斜体（*`constant width bold`*）

  表示应该由用户输入的值或根据上下文确定的值替换的文本。

使用示例代码

本书的补充资料（示例代码、练习等）可以从 https://github.com/cfjedimaster/DataStorageBook 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，那么你可以把它用在你的程序或文档中。除非你复制了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书示例的光盘则需要获得许可，引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Client-Side Data Storage* by Raymond Camden (O'Reilly).

Copyright 2016 Raymond Camden, 978-1-491-93511-8."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于公司企业、政府机构、教育机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。

用户可通过一个功能完备的数据库检索系统并访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他数百家出版社的成千上万的图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网页地址是：

http://shop.oreilly.com/product/0636920043676.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

一如既往地感谢我的妻子。你激励着我。你让我心态平和。你让我开怀大笑。我爱你。

电子书

扫描如下二维码，即可购买本书电子版。

第 1 章

客户端数据存储概述

过去十年中，浏览器已经发展成为一个强大的工具。这是一个缓慢的过程，伴随着许多成长之痛。现在，增强型布局控件、3D 图形和游戏，甚至是音乐都可以在小而古老的浏览器中实现。客户端数据存储是一个更加令人兴奋的特性，虽然相比之下，它没那么华丽（没有别的意思）。但我们为什么这样说呢？

浏览 Web 的“经典”过程从一开始就没有变过：浏览器请求一个 URL，Web 服务器返回请求的内容，然后浏览器请求更多的内容，而服务器则返回更多的内容。

当然，你可以引入 JavaScript 和 AJAX，让情况更复杂一些。但是，即使是在精心设计的 Web 2.0 应用程序中，浏览器还是会一次又一次地向服务器请求信息。之所以会这样，是因为浏览器（似乎）很健忘。它知道的所有东西都必须从服务器习得。

虽然一般而言确实如此，但这忽视了一个功能强大的替代方案：将数据存储在浏览器中，让它可以跳过向服务器请求信息的过程，而只从用户的本地机器上获取数据。它甚至还可以操作那些数据，用于任何合理的用途。数据可以稍后被发回服务器用于更新。

总之，这让浏览器可以拥有如下能力。

- 直接访问数据。虽然使用 AJAX 获取数据的速度通常已经快了很多，但将数据存储在本地机器上会让数据访问速度更快。
- 节省网络流量。浏览器获取一次数据，只要有用就一直保存着，而不必不断地从服务器获取数据。这能够减轻服务器的压力。
- 减轻服务器的压力。如果服务器不断地响应请求，并从数据库服务器获取数据，那么服务器会负担过重。减少请求次数，可以减少服务器的工作量。

- 最后，数据存储在本地，这使创建完全离线的应用程序变得更加可行。

当然，并非一切都如此美好。将数据转移到浏览器也有以下几点不足。

- 没有任何同步支持。设想一下，你已经将数据从服务器复制到了浏览器。如何处理数据同步呢？如果出现冲突会怎样？本书所谈及的核心技术，没有一项支持任何有关同步处理的概念。不过，你会发现，像 PouchDB（http://www.pouchdb.com）这样的库内置了同步功能。
- 存储限制模糊。作为开发人员，我们讨厌模糊。我们希望准确地知道可以使用多少资源。遗憾的是，对于本书将要谈及的许多技术而言，这些限制（以及打破限制的后果）有点模糊。
- 最后，虽然本书谈及的技术功能非常强大，但它们并不能取代纯正的数据库服务器。数据库服务器针对处理大量数据的任务进行了特别仔细的优化，并提供了查找数据的方法。本书将要谈及的方案无疑能够存储数据，但它们并不像一个嵌入式 Oracle 服务器。（虽然这可能是一件好事。）

本书将讨论各种客户端存储技术。对于每一种技术，本书还会清楚公正地讨论它们实际上得到了多大程度的支持。你将会看到 API 示例以及演示程序，它们可以帮助你学习如何使用 API。最后，我们将看几个旨在简化客户端存储的库。做好准备，让我们浏览一些更为实用的 Web 特性，这将是一段有趣但时而艰难的旅途。

第 2 章

使用Cookie

2.1 真的要讨论Cookie吗

在一本有关现代 Web 开发的书里讨论 Cookie，我真是有点不好意思，但是，这是开发人员如今可以使用的最古老、最稳定的客户端存储形式。当然，Cookie 不是最好的方法，我几乎从来不建议使用它，但它是一种选择，在将来的某个时候，你也许不得不使用（或修改）应用了 Cookie 的代码。

Cookie 于 1994 年在 Netscape 浏览器的一个 Beta 版本中被引入。它通过随 HTTP 请求和响应一起发送的 HTTP header 值发挥作用。众所周知，每当浏览器请求一个资源，就会有一组 header 随请求一起发送。那些 header 包含各种类型的数据，其中包括有关浏览器的信息以及它需要的数据形式。反过来，服务器也会往回发送 header。基本上，每次你看到浏览器渲染一个 Web 页面，就有一组你看不到的 header 被发送。（当然，你可以使用浏览器工具查看它们。它们并没有被隐藏得“无法看到”，只是在默认情况下看不到。）

Cookie 使用 HTTP header 发送，具体来说是名为“Cookie”的 HTTP header，由浏览器发送到服务器，又从服务器发送到浏览器。你会发现这里有个问题。如果使用客户端存储的一个好处是不用通过网络发送数据，那么来回发送 Cookie 不是反其道而行之吗？完全正确。这就是我通常不建议使用 Cookie 的另一个原因。

默认情况下，浏览器没有限制它可以拥有的 Cookie 数量。以前，每个域名最多只能有 20 个 Cookie，但如今的浏览器似乎已经去掉了这个限制。（顺便说一句，我曾经在 Chrome 浏览器中设置了 400 多个 Cookie，它仍然可以正常运行。不过，当它发出请求时，Web 服务

器开始抛出错误。因此，这种情况是 Web 服务器有限制的问题，而不是浏览器本身有限制。但请不要使用 400 个 Cookie。）研究（或者说 Google 搜索结果）表明，每个域名 50 个、大小总计 4KB 的 Cookie 是安全的，不过这存储不了太多 Cookie 值，会影响它们的实际应用。

Cookie 对应唯一的域名。这意味着在 foo.com 上设置的 Cookie 值不能用于 goo.com。这样很好，因为你不会希望其他网站影响你在自己的网站上使用 Cookie。Cookie 也可以对应唯一的子域名。例如，app.foo.com 是 Foo 网站的一个独立的子域名。你可以创建只有 app.foo.com 可以读取的 Cookie，也可以创建 www.foo.com 和 app.foo.com 都可以读取的 Cookie。

更复杂的做法是创建只对特定路径有效的 Cookie。所以，你可能希望创建只有 foo.com/app 可见的 Cookie。

最后，你可以创建只对网站的安全（HTTPS）版本有效的 Cookie。显然，选用哪种方案取决于应用程序的用途，以及你认为哪里需要 Cookie 值。

除了设置 Cookie 出现的地方，还可以指定 Cookie 的有效时间。对此，你有以下几个选项：

- 只在当前会话期间存在的 Cookie（从根本上说是直到浏览器关闭）；
- 永远存在的 Cookie；
- 存在特定时长的 Cookie；
- 特定时间点之后失效的 Cookie。

2.2 使用Cookie

Cookie 没有 API。要使用 Cookie，只需要在代码中访问 `document.cookie` 对象。例如，可以像下面这样创建一个 Cookie。

```
document.cookie = "nameOfCookie=value";
```

上例创建了一个名为 `nameOfCookie` 的 Cookie，并将它的值定义为 `value`。你可以使用 `"name=value"` 同时定义名称和值。以下是一个实例。

```
document.cookie = "name=Raymond";
```

在上面的例子中，我简单设置了一个名为 `name`、值为 `Raymond` 的 Cookie。值必须符合 URL 编码规则，这意味着如果想动态定义 Cookie，那么就需要使用类似 `encodeURIComponent` 的辅助函数，如下所示。

```
name = "Raymond Camden";
document.cookie = "name=" + encodeURIComponent(name);
```

目前为止，一切顺利。但这里就是事情变得有点古怪的地方。你可能想知道如何设置多个 Cookie。这能够通过直接对 document.cookie 进行多次设置实现，如下所示。

```
document.cookie = "name=Raymond";
document.cookie = "age=43";
```

这段示例代码实际上创建了两个 Cookie，而不是一个。我觉得这完全不符合逻辑，但是我们必须适应这种定义方式。

Cookie 就是这样创建并赋值的，但是，我提到过的所有其他元数据（像定义 Cookie 在哪里可见以及它存在多长时间）呢？在 Cookie 值后面使用一个分号可以追加元数据。下面是一个例子。

```
document.cookie = "name=Raymond; expires=Fri, 31 Dec 9999 23:59:59 GMT";
```

这个例子指明了 Cookie 何时过期。我们可以进一步扩展，指定该 Cookie 只对一个子域名有效。

```
document.cookie = "name=Raymond; expires=Fri, 31 Dec 9999 23:59:59 GMT;
domain=app.foo.com";
```

你已经了解了如何设置 Cookie。当你不这样指定元数据时，Cookie 默认只对当前域名的当前路径有效（你可能不希望这样），有效期是当前会话。

2.2.1 读取Cookie

读取 Cookie 多少简单一些——这取决于你对字符串解析的习惯程度。没有 API 可以用来获取“一个”Cookie。不过，你只需要简单地读取 document.cookie 就可以了。这样，你就可以获取特定网站的所有 Cookie。以下是从 CNN 获取的 document.cookie 值。

```
"_cb_ls=1;
_chartbeat2=Dlxk2YDHxyg1BXCry6.1426601000831.1439508384927.0000000000000001;
Akamai_AnalyticsMetrics_clientId=89E881222E0BD593DF2468758F328F689C36BAC1;
octowebstatid=16ppgnhrso5f2frjuvq5; ug=55cd27810eb00b0a3c6ac33c7d05339d; ugs=1;
__CG=u%3A2449373858398994400%2Cs%3A72001958%2Ct%3A1439508379253%2Cc%3A1%2Ck%3
Awww.cnn.com/19/19/54%2Cf%3A0%2Ci%3A0; __CT_Data=gpv=10;
__gads=ID=3d001e8bba3c7c6d:T=1426601001:S=ALNI_MYWNYv1SRt0tx7LQ2AzdSESOBygNA;
__vrf=1439508379290061VnNeHVWPiIjkcWMeUjRWpppwsPktE;
grvinsights=a5a942f8e7c604d573496053d63f590c; optimizelyBuckets=%7B%7D;
optimizelyEndUserId=oeu1426600996913r0.5135214943438768;
optimizelySegments=%7B%22170962340%22%3A%22false%22%2C%22171657961%22%3A%22
safari%22%2C%22172148679%22%3A%22none%22%2C%22172265329%22%3A%22direct%22%7D;
RT=sl=1&ss=1439508375405&tt=9387&obo=0&bcn=%2F%2F36f11e49.mpstat.us%2F&sh=1439
508384794%3D1%3A0%3A9387&dm=cnn.com&si=5be398d8-bb51-42ea-8128-6d4251e47ada;
 s_cc=true;s_fid=5324AC5D0F8323AB-3F26DEB602CBB276; s_ppv=13; s_sq=%5B
%5BB%5D%5D;s_vi=[CS]v1|2A841A13051D0B97-400001280000490C[CE]; tosAgreed=true"
```

是不是看起来一团糟？完全正确。读取一个 Cookie 就意味着将字符串解析成多个由分号分隔的部分。另外还要注意，你无法访问任何元数据。通过 `document.cookie` 值无法获取这类信息。虽然字符串解析并不是很难，但实际上，你不需要那样做。在本章的末尾，我会介绍一个优秀而小巧的库，它让你可以更轻松地使用 Cookie。

2.2.2 删除Cookie

要删除 Cookie，只需要将其过期时间设置成过去的时间，如下所示。

```
document.cookie = "name=Raymond; expires=Thu, 01 Jan 1970 00:00:00 GMT";
```

从技术上讲，这个时间值无关紧要，但名称必须与你想要删除的 Cookie 名称一致。

2.3 演示程序

你已经了解了使用 Cookie 的基本知识，让我们看一个简单的演示程序。我之前说过，你也许不希望自己构建解析 Cookie 的代码。相反，你应该从许多已有的库中选择一种来用。在演示程序中，我们将使用一个由 Mozilla 开发者网络（Mozilla Developer Network，MDN）提供的简单（并且优秀）的免费库。你可以通过 https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie 找到这些代码（及更多有关 Cookie 的信息）。该库包含如下方法。

- `getItem`：获取 Cookie
- `setItem`：设置 Cookie（包括有效期、路径、域名，等等）
- `removeItem`：删除 Cookie
- `hasItem`：检查 Cookie 是否存在
- `keys`：返回所有 Cookie 的名称

MDN 提供的代码已经被保存在文件 cookies.js 中。该文件会和本书的其他代码一起提供。第一个例子（test1.html）只是简单地使用 Cookie 统计你访问网站的次数（请见示例 2-1）。

顺便说一下，本书所有的示例都假定（并需要）你是在本地 Web 服务器上运行它们，而不只是在浏览器中打开文件。如果你没有在本地安装 Web 服务器，则可以考虑使用一个类似 httpster（https://simbco.github.io/httpster/）的简单工具，快速搭建一个供开发使用的 Web 服务器。（但答应我，稍后安装一个合适的 Web 服务器。你是 Web 开发人员，对吧？）

示例 2-1 test1.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Cookie Test One</title>
    <meta name="description" content="">
```

```
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script type="text/javascript" src="cookies.js"></script>
</head>

<body>

    <div id="resultDiv"></div>

    <script>
    $(document).ready(function() {

        //初始值
        var visited = 0;

        //检查Cookie是否存在……
        if(docCookies.hasItem("visited")) {
            //获取Cookie
            visited = docCookies.getItem("visited");
        }

        visited++;

        //更新
        docCookies.setItem("visited", visited);

        $("#resultDiv").text("You have visited this site "+ visited +
             " times.");

    });
    </script>

</body>
</html>
```

在代码开始的部分，你可以看到一个相当典型的 `document.ready` 块，它来自简单易用的 jQuery 库。首先，我为变量 `visited` 设置了一个初始值。如果 Cookie 解析库通过检测发现名为 `visited` 的 Cookie 已经存在，那么我会使用该 Cookie 的值更新这个变量。然后，将变量的值加 1 再存回到 Cookie 中，由浏览器显示出来。在默认情况下，该 Cookie 只在当前会话期间有效，因此，我们在 test2.html（示例 2-2）中对此进行了改进。

示例 2-2　test2.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Cookie Test Two</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
```

```
    <script type="text/javascript" src="cookies.js"></script>
</head>
<body>

    <div id="resultDiv"></div>

    <script>
    $(document).ready(function() {

        //初始值
        var visited = 0;

        //检查Cookie是否存在……
        if(docCookies.hasItem("visited2")) {
            //获取Cookie
            visited = docCookies.getItem("visited2");
        }

        visited++;

        //更新
        docCookies.setItem("visited2", visited, Infinity);

        $("#resultDiv").text("You have visited this site "+ visited +
            " times.");

    });
    </script>

</body>
</html>
```

在这个版本中，我们作了两处修改。首先，Cookie 的名称被改为 `visited2`。通常，我会使用以前用过的名称，但我们希望在测试的过程中区分这两个 Cookie。第二个变化是修改了 `setItem` 方法的调用方式。我们使用值 `Infinity` 作为有效期。这样，创建的 Cookie 会一直存在。现在，页面会准确地反映特定用户访问网站的次数，而不管用户是否是在同一个浏览器会话期间进行访问。

到目前为止，演示程序还算非常简单，所以，让我们提高一下难度。我们可以使用 Cookie 记住用户最后一次访问网站的时间。根据这个时间，我们可以做以下事情：

- 问候以前从来没有访问过该网站的新用户；
- 为隔了一段时间再来访问网站的用户提供重要的信息，比如介绍很酷的新功能；
- 简单地欢迎经常访问网站的用户。

让我们看一下示例 2-3 的代码，随后我会向你解释它的执行过程。

示例 2-3　test3.html

```
<!DOCTYPE html>
<html>
```

```
<head>
    <meta charset="utf-8">
    <title>Cookie Test Three</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script type="text/javascript" src="cookies.js"></script>
</head>
<body>

    <div id="resultDiv"></div>

    <script>
    $(document).ready(function() {

        var $resultDiv = $("#resultDiv");

        //这是一名新用户吗?
        var newUser = true;
        //从最后一次访问到现在有多少天
        var daysSinceLastVisit;

        //检查Cookie是否存在……
        if(docCookies.hasItem("lastVisit")) {
            newUser = false;

             //计算出最后一次访问距离现在多久了
             var lastVisit = docCookies.getItem("lastVisit");
             var now = new Date();
             var lastVisitDate = new Date(lastVisit);
             //参见http://stackoverflow.com/a/3224854/52160
             var timeDiff = Math.abs(now.getTime() - lastVisitDate.getTime());
             var daysSinceLastVisit = Math.ceil(timeDiff / (1000 * 3600 * 24));
         }

         //将lastVisit设为当前时间
         docCookies.setItem("lastVisit", new Date(), Infinity);

         if(newUser) {
             $resultDiv.text("Welcome to the site!");
         } else if(daysSinceLastVisit > 20) {
             $resultDiv.text("Welcome back to the site!");
         } else {
             $resultDiv.text("Welcome good user!");
         }

    });
    </script>

</body>
</html>
```

我们首先设置了两个变量——newUser 和 daysSinceLastVisit。前者是布尔类型，我们可以

通过它确定用户是否是新用户，而后者将报告用户最后一次访问距离现在的天数。

如果名为 `lastVisit` 的 Cookie 已经存在，那么我们就获取它的值，然后以这个值为基础创建一个 `Date` 类型的变量。然后，我们就可以使用一些简单的数学运算，计算出用户最后一次访问到现在有多少天了。

接下来，我们将 `lastVisit` 的 Cookie 值设为当前时间。

这就是核心逻辑。然后，我们只需要根据上述三种状态中的一种发出一条消息。在这个演示程序中，超过 20 天就认为用户最后一次访问距离现在太久了。显然，这个值是任意的。

2.4 使用开发者工具查看Cookie

如今的浏览器都提供了非常优秀的开发者工具，让你可以更轻松地查看 Cookie 的使用情况。在 Firefox 中，你需要启用存储选项卡（Storage，默认情况下可能不显示）才能看到 Cookie。一旦启用，你就可以查看当前的 Cookie 了（如图 2-1 所示）。Firefox 不允许修改 Cookie，只能查看。

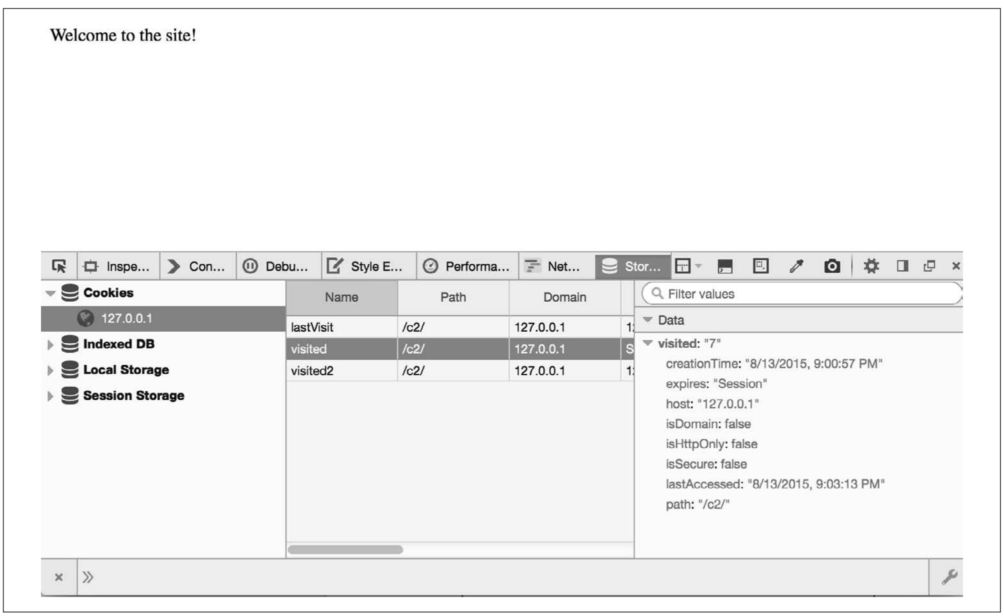


图 2-1：Firefox 开发者工具中显示的 Cookie

Chrome 在资源选项卡（Resources）中显示站点当前的 Cookie（如图 2-2 所示）。你可以在这里删除 Cookie，但不能编辑。

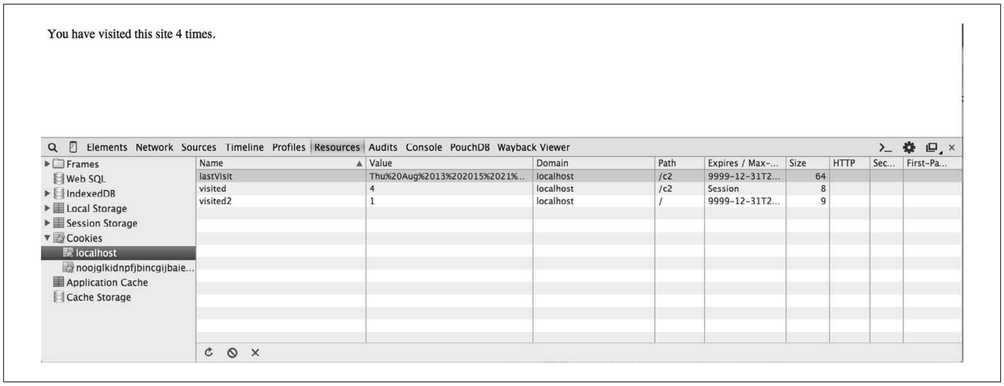


图 2-2：Chrome 中显示的 Cookie

2.5　浏览器支持和使用建议

CanIUse.com 是核实浏览器特性支持情况的最佳资源，但它没有提供关于 Cookie 的报告，这是因为 Cookie 很久之前就已经获得了 100% 的支持。不过，仅仅浏览器支持这项特性并不能保证它可以正常使用。许多人都因为对 Cookie 存有戒心而屏蔽了它们。

至于使用建议，就像我在本章开头所说的那样，我的建议是不使用 Cookie，但如果你一定要用，那么也以简单为好。可以用它们存储用户偏好和基本信息（姓名、年龄等）。举个实际的例子。我使用 WordPress 写博客。当我上传图片时，WordPress 总会在将图片添加到博文时，询问我是否希望包含图片链接。我几乎总是更改这个特定的表单域，告诉它我不需要链接。这时，就可以使用 Cookie 记住我的选择，那样我就不必在写博客的过程中不断地更改这个表单域。这是一个很简单的例子，却是我几乎每天都会碰到的事。最后，如果你认为有些东西是服务器也应该知道的，那么就可以使用 Cookie 确保服务器看到的是同样的值。

第3章

使用Web存储

3.1 Web存储/本地存储

我们大多数人所说的本地存储技术，其正式名称为 Web *存储*。在本书将要讨论的所有客户端技术中，Web 存储 API 可能是学习周期最短的，也是最容易学会的。该 API 主要通过键设置和检索简单的值。例如，存储键 `name` 的值 `Ray`，或者存储键 `age` 的值 `43`。它不支持复杂数据（比如数组或对象），但你可以把这样的值预先编码成 JSON，然后再存储。（显然，在检索时需要解码。）

Web 存储有两个版本：*本地存储*（Local Storage）和*会话存储*（Session Storage）。两者使用完全相同的 API，但本地存储会持久存在（或者直到用户清除），而会话存储只要浏览器关闭就会消失。因为大多数人都使用持久化版本，所以大多数开发人员使用（和谈论）的都是本地存储。Web 存储 API 官方规范的网址为 http://www.w3.org/TR/webstorage。

和 Cookie（及本书涉及的其他技术）类似，Web 存储是与域名一一对应的。和 Cookie 不同的是，无法让 app.foo.com 使用 www.foo.com 存储的数据。（可以借助 iframe 变通实现，但比较复杂，这里暂且不讲。）从根本上说，这意味着 foo.com 和 goo.com 都可以安全地使用名为 `name` 的 Web 存储键——它们不会冲突。

Web 存储的限制没有一定之规，但一般为 5~10MB。一般来说，这应该够用了，除非你要存储大数据包，而我（通常，但并不总是）不建议这样做。如果超出了限制，则 Chrome、Firefox 和 Safari 浏览器都会报告一个你可以在代码中进行处理的错误。但遗憾的是，Internet Explorer 11 和 Edge（在本书写作时）什么也不会做。

3.2 使用Web存储

Web 存储 API 有如下 4 个简单的方法（本章所有的演示程序都会使用本地存储，但请记住，会话存储版本的用法完全相同）。

- `localStorage.setItem`：设置特定键的值
- `localStorage.getItem`：检索特定键的值
- `localStorage.removeItem`：删除键及与其关联的值
- `localStorage.clear`：删除所有的键 / 值对（但只限于发出请求的特定域名）

虽然 Web 存储提供了 API，但仍然可以像对待简单的 JavaScript 对象那样处理数据。例如，下面的语句：

```
localStorage.setItem["something"] = 1
```

会写入 Web 存储，而语句：

```
console.log(localStorage["something"]);
```

会读取存储。虽然这可以正常运行，但为了一致起见，我一般建议使用 API 方法。

有一件事你必须非常小心，那就是在 Web 存储中存储什么数据。Web 存储仅支持字符串数据。这有时会引起混淆。考虑下面这段代码。

```
var names = ["Ray", "Jeanne"];
localStorage.setItem("names", names);
```

这段代码可以正常运行。不过，它会存储数组的字符串版本，而不是数组本身。也就是说，如果你稍后调用 `localStorage.getItem("names")`，那么将得到字符串 `"Ray,Jeanne"`，而不是期望的数组。

幸运的是，有一种相当简单的变通方案：JSON 编码。将复杂的数据转换成 JSON，然后在获取值的时候通过解码进行还原，就可以轻松地将复杂的数据存入 Web 存储了。下面是上述代码片段的修正版本，它使用了现代浏览器提供的 JSON 对象。（对于比较老的浏览器，有许多库可以让你添加这种支持。）

```
var names = ["Ray", "Jeanne"];
localStorage.setItem("names", JSON.stringify(names));
```

将值重新读到数组里也非常简单：

```
var storedNames = JSON.parse(localStorarge.getItem("names"));
```

为了确保这种方法有效，你需要记住什么键存储了什么类型的值。因此，务必随时记录。

你可以使用这样一种命名系统，就是将所有值为 JSON 编码的键都加上前缀 `js` 或 `json`。

如你所见，该 API 非常简单。下面让我们看几个演示程序。

3.3 演示程序

第一个演示程序非常简单，而且有点老套。我们将使用 Web 存储记录你访问页面的次数（示例 3-1）。上一章讲过，要使用一个合适的 Web 服务器来测试，这里再次提醒。不要仅仅通过双击打开文件，而是要在本地 Web 服务器上运行它。

示例 3-1 test1.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>WebStorage Test One</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>
<body>

    <div id="resultDiv"></div>

    <script>
    $(document).ready(function() {

        if(window.localStorage) {
            var numHits = localStorage.getItem("numHits");
            if(!numHits) numHits=0;
            else numHits = parseInt(numHits, 10);
            numHits++;
            localStorage.setItem("numHits",numHits);
            $("#resultDiv").text("You have visited this site " +
                numHits +" times.");
        }
    });
    </script>

</body>
</html>
```

示例代码包含一个简单的 `div` 块，我们将用它显示你访问网站的次数。JavaScript 代码首先检查 `window.localStorage` 是否存在。虽然 Web 存储已经获得了相当好的支持（在本章末尾你会看到），但检查并确保浏览器支持这一特性只需要很少的代码。接下来，获取 `numHits` 键的值。如果什么也没取到，那就默认其值为 `0`。否则，使用 `parseInt` 将字符串值转换成一个正确的数值。记住，Web 存储将一切数据作为字符串存储，甚至数值也是如此。

接下来，我们增加计数器的值，并重新存入 Web 存储，然后将结果显示在屏幕上（如图 3-1 所示）。我们原本可以做得更好，即显示“1 time”而不是“1 times”，但就我们的目的而言，那过于复杂了。



图 3-1：演示程序多次运行之后

如前所述，因为 Web 存储 API 的会话存储版本并没有什么不同之处，所以我们没有涉及，只是将相关的代码包含在一个名为 test1_session.html 的文件中。那是示例 3-1 的一个修正版本，简单地说明了如何使用 `sessionStorage` 对象，而不是 `localStorage`。

现在，让我们更进一步。接下来的演示程序将展示一些真正实用的东西。你是否曾经在填写表单时，意外关掉了浏览器页签？或者表单是在一个需要登录信息的网站上，而在你填完表单之前登录过期了？在示例 3-2 中，我们将使用 Web 存储保存表单数据的一个副本，这样数据就不会丢失了。当表单真正填写完成后，我们还需要删除表单数据。

示例 3-2　test2.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>WebStorage Test Two</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>

</head>
<body>


    <form id="myForm">
```

```
    <p>
    Your Name:
    <input type="text" id="name" name="name">
    </p>

    <p>
    Your Age:
    <input type="number" id="age" name="age">
    </p>

    <p>
    Your Email:
    <input type="email" id="email" name="email">
    </p>

    <p>
        <input type="submit">
    </p>

</form>

<script>
$(document).ready(function() {

    if(window.localStorage) {

        //如果有数据,则获取并预设
        if(localStorage.getItem("personForm")) {
            var person =
            JSON.parse(localStorage.getItem("personForm"));
            $("#name").val(person.name);
            $("#age").val(person.age);
            $("#email").val(person.email);
            console.log("restored from storage");
        }

        //监听所有<input>字段及其输入事件
        $("input").on("input", function(e) {
            var name = $("#name").val();
            var age = $("#age").val();
            var email = $("#email").val();
            var person = {"name":name, "age":age, "email":email};
            localStorage.setItem("personForm",
                JSON.stringify(person));
            console.log("stored the form...");
        });

        //表单处理器应该清除存储
        $("#myForm").on("submit",function(e) {
            localStorage.removeItem("personForm");
            return true;
        });
    }
```

```
    });
    </script>

</body>
</html>
```

文件上半部分正是我们要存留的表单。它有三个输入字段和一个提交按钮。（注意：`<form>` 标签并没有 `action` 值，这是因为我们并不是真要构建一个表单处理器。）

在 JavaScript 部分，我们要再一次确保浏览器支持 Web 存储，然后才开始处理业务。

首先要做的是查看 Web 存储中是否存在表单数据。我们获取 `personForm` 键的值，如果该值存在，则它会是一个 JSON 编码的对象。取得该值并解码之后，我们就可以使用已有的数据逐个更新表单字段。因为它们都是普通的文本字段，所以这个过程并不难。但显然，只要稍微多做一些工作，你就可以支持 select、checkbox 和 radio 字段。

接下来，我们在表单的输入字段上添加一个事件监听器。输入事件每次被触发都意味着输入字段的内容发生了变化。我们获取这些字段的值，把它们存储在一个简单的对象中，然后将 JSON 编码版本存入 Web 存储。

最后，在用户提交表单时（如图 3-2 所示），我们就不再需要保存表单数据的副本了。（虽然从技术上讲确实如此，但如果这是一个人们经常使用的表单，那么你可能会希望保存某些字段。）

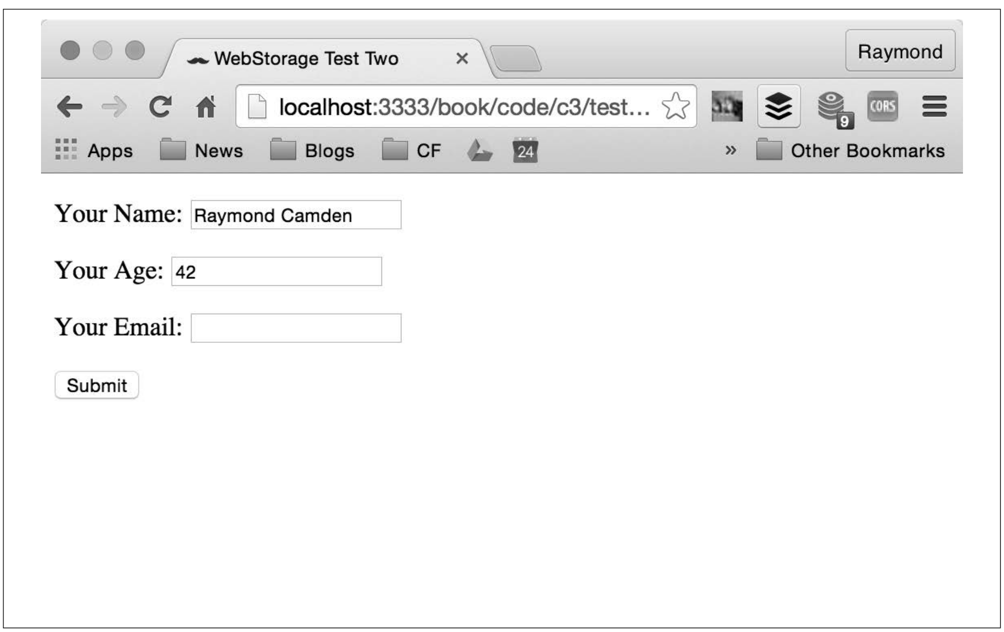


图 3-2：自动加载部分样例数据的表单

3.4 监听存储变化

我们将要探讨的最后一项特性是存储事件。该特性有一点奇怪，而且可能不是你需要担心的问题，但绝对值得讨论。顾名思义，存储事件是存储（包括本地存储和会话存储）被修改时抛出的事件。让我们看一个简单的例子（示例 3-3）。

示例 3-3 test3.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>WebStorage Event Test</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>

</head>
<body>

    <form id="myForm">

        <p>
        Test Value:
        <input type="text" id="test">
        </p>

    </form>

    <script>
    $(document).ready(function() {

        if(window.localStorage) {

            if(localStorage.getItem("testValue")) {
                $("#test").val(localStorage.getItem("testValue"));
            }

            //监听所有<input>字段及其输入事件
            $("input").on("input", function(e) {
                var test = $("#test").val();
                localStorage.setItem("testValue", test);
                console.log("stored the test value.");
            });

            $(window).on("storage", function(e) {
                console.log("storage event fired");
                console.dir(e);
            });

        }
```

```
    });
    </script>

</body>
</html>
```

在页面上方，你会看到一个简单的表单，它有一个 id 为 `name` 的文本字段。可以看到，JavaScript 代码和上一个演示程序有点类似。页面加载时，我们首先会查找先前存在的值，并用它设置该字段。然后，一个普通的输入修改处理器会监听字段的变化，并存储它们。注意，该演示程序使用 `console.log` 方法记录保存日志。

接下来，我们有一个新的事件监听器。因为存储事件是在 `window` 对象上触发的，所以我们就监听该对象。稍后，我们将进一步讨论事件内容。不过现在，让我们继续，用浏览器打开文件并做一些测试。简单输入一些内容，并修改几次。你会看到类似图 3-3 所示的内容。

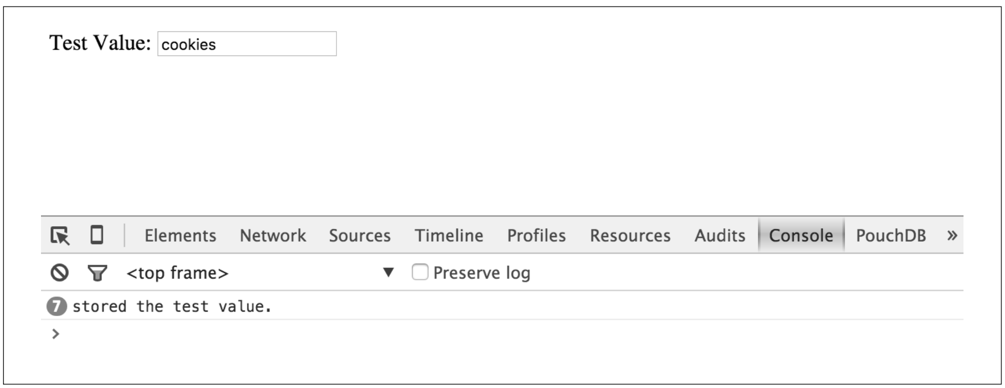


图 3-3：存储事件在哪里被触发?

注意到什么异常了吗？控制台上有多条关于字段值存储的日志信息，但没有任何关于存储事件本身的信息！这里到底发生了什么？

好吧，事实是，存储事件只有在浏览器的另一个实例修改存储时，才会被触发。怎么才会出现那种情况呢？只要另外打开一个页签，输入相同的 URL。在新打开的页签中修改字段值，然后返回原来的页签，你就会看到关于事件本身的消息了。这里的情况是，存储事件让你知道其他代码修改了存储。

请注意，在图 3-4 中，事件包含两个有趣的值：`oldValue` 和 `newValue`。你可能已经猜到，事件正在报告原始值以及修改后的值。

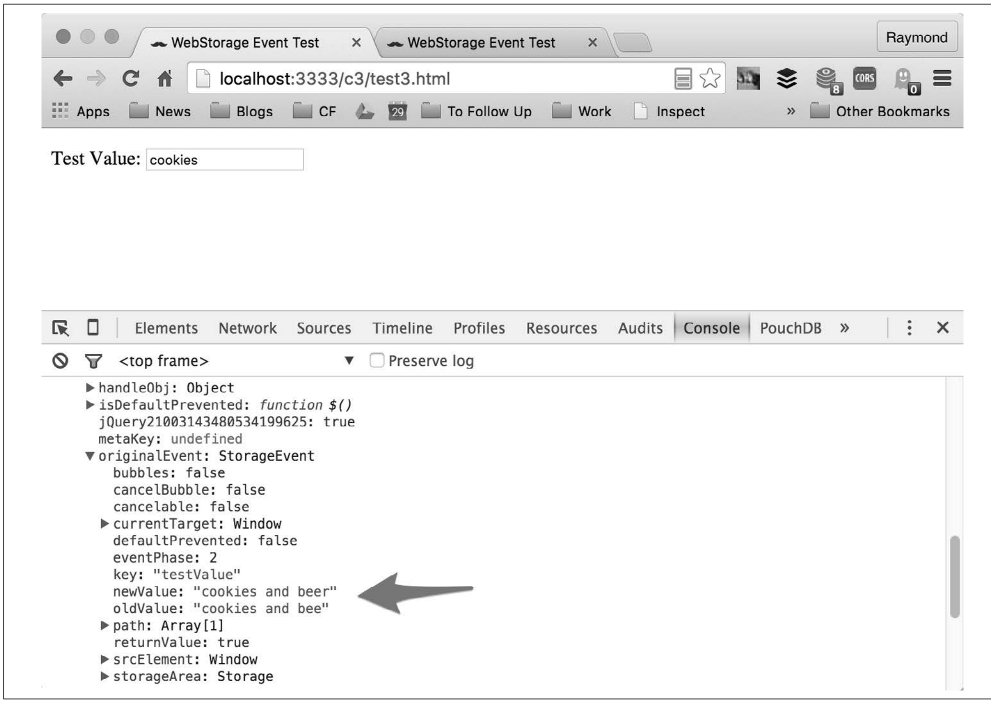


图 3-4：存储事件被触发！

现在的问题是，你要做什么？实际上，如何处理变化取决于你的应用程序。你可能希望提示用户，询问他们是想要接受修改，还是保持当前的版本。显然，这已经太晚了，因为存储系统已经变了。但是，你可以获取表单的值并更新存储。当然，其他页签接着就会看到同样的警告信息。因此，提示用户或许是一个坏主意。更简单的做法可能是仅更新表单字段，让它包含最新的值。示例 3-4 说明了这种方法。（由于我们只修改了存储事件，因此程序清单只包含这部分代码。）

示例 3-4　test4.html（片段）

```
$(window).on("storage", function(e) {
    console.log("storage event fired");
    $("#test").val(e.originalEvent.newValue);
});
```

可以看到，我们只是使用新值更新了表单字段。我们原本也可以使用 `localStorage.getItem("testValue")` 方法，但由于事件已经包含了这个值，因此直接使用它也是合理的。

3.5　使用开发者工具查看Web存储

对于 Web 存储的使用，Firefox 和 Chrome 都在各自的浏览器开发者工具中提供了很好的支持。从图 3-5 中，你可以看到 Firefox 如何显示演示程序的本地存储数据。

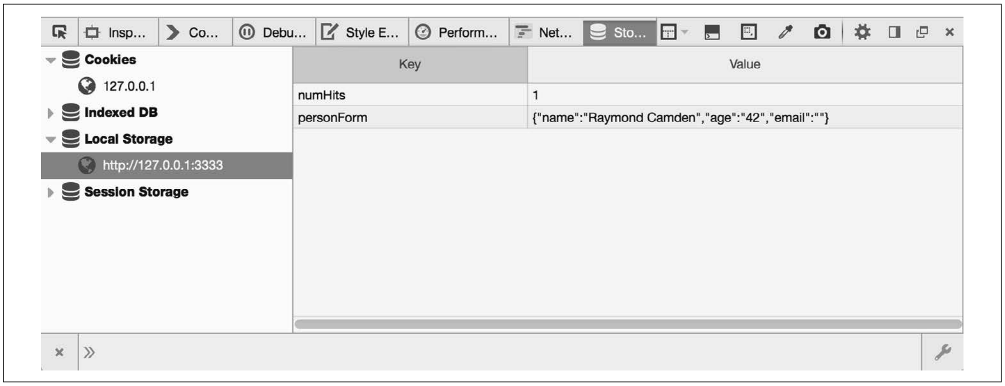


图 3-5：Firefox 开发者工具的 Web 存储视图

如果你觉得这不够直观，则可以点击一个值，查看其详细视图。Firefox 能够识别出 `personForm` 中的 JSON 字符串，并且会以更友好的方式显示出来（如图 3-6 所示）。

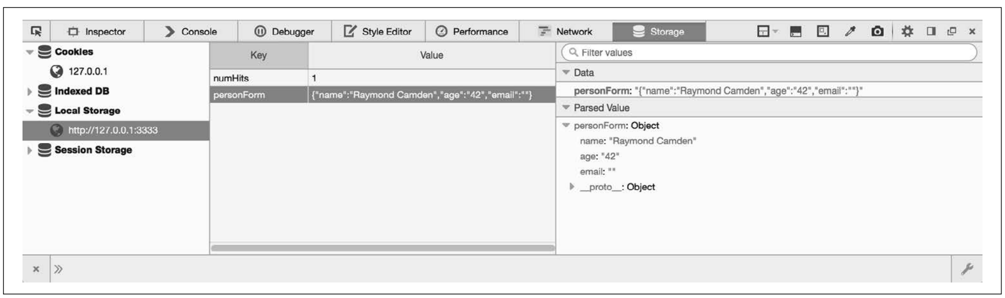


图 3-6：Web 存储值的详细视图

在 Firefox 中，Web 存储的值无法编辑或删除。不过，不要忘了，你可以通过控制台选项卡直接操作那些值。

在 Chrome 中，你可以在资源选项卡中查看 Web 存储的值（如图 3-7 所示）。

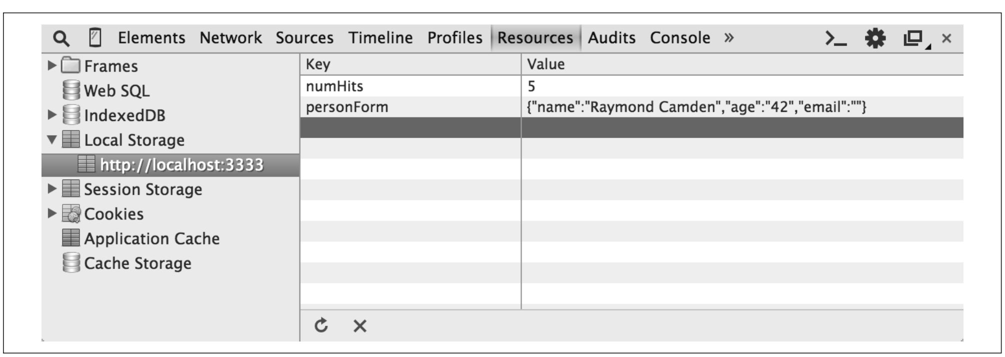


图 3-7：Chrome 开发者工具的 Web 存储视图

和 Firefox 不同，你可以双击一个值并手动进行编辑。你还可以使用界面底部的 X 图标删除某条记录。

3.6　浏览器支持和使用建议

Web 存储的浏览器支持情况怎么样呢？图 3-8 是 CanIUse.com 给出的答案。

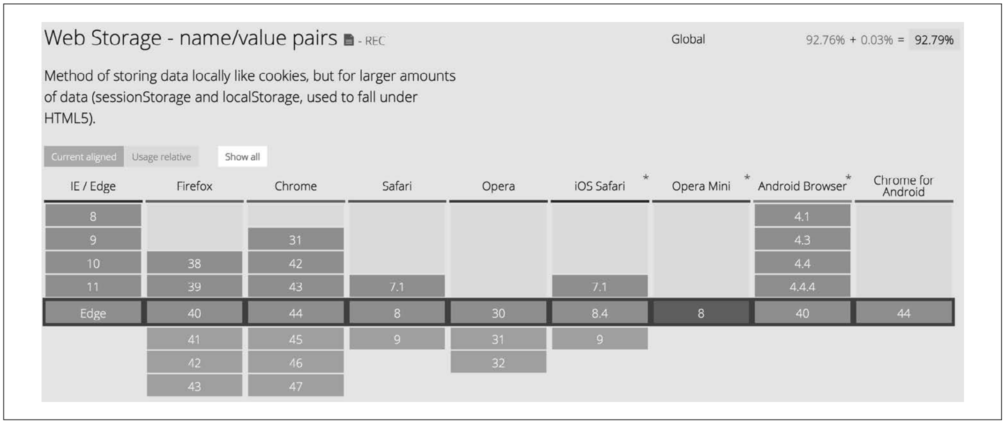


图 3-8：CanIUse.com

浏览器提供了非常好的支持。就像你在演示程序中看到的那样，检查浏览器是否支持该特性并使用它增强页面功能相当简单。在这两个演示程序中，即使浏览器不支持 Web 存储，也不会造成什么破坏。一般来说，这是我们在构建 Web 页面时应该采用的方式！

至于使用建议，我认为它适合存储一些简单的信息，用户偏好就是一个很好的例子，还有一些基本信息，比如用户的姓名、年龄等，或许还可以存储一个网站“最受欢迎”内容的列表。当然，这些信息也可以被存储在服务器端，但是，由于它们是特定于用户的，对于站点本身而言并不重要，因此把它们存储在客户端可能更合适。

第 4 章

使用IndexedDB

4.1 欢迎来到深度数据时代

到目前为止，我们所使用的客户端数据存储系统本身都比较简单小巧。现在，是时候更深入一些，使用一个大型存储系统了。IndexedDB 是一个功能强大且高度灵活的存储系统。你可以使用它在用户浏览器中存储你希望存储的任何数据。不过，出色的功能和灵活性致使其 API 不像 Web 存储那么友好。你还会发现，移动端浏览器对 IndexedDB 的支持还不是很好，即使支持，其实现方式也很糟糕（iOS 8 对 IndexedDB 的支持尤其差，你最好当它不支持）。不过，IndexedDB 将来可能会成为在客户端存储大量数据的标准方法。要了解更多信息，可以查看 IndexedDB 规范，网址为 http://www.w3.org/TR/IndexedDB/。这会是一次令人兴奋的阅读体验。（真的！）

和本书之前所介绍的其他客户端存储系统一样，IndexedDB 是和域名一一对应的。通常，浏览器都不会明确定义存储限制，即使定义了，往往也非常大。可以说，基本上没有限制。但是，当存储空间低于某个标准时，浏览器就会开始清理其他 IndexedDB 实例。和大多数“持久化”系统一样，从根本上说，存储在浏览器中的任何数据都不是永远存在的，但客户端数据存储所带来的好处，即使是半持久化，也值得我们这样做。

4.2 IndexedDB关键术语

在开始讲解代码之前，让我们先看几个重要的 IndexedDB 术语。

数据库

IndexedDB 的最上层是数据库的概念。如果你曾经在服务器端 Web 应用程序中使用过数据库，那么应该已经熟悉这个概念了。从本质上说，数据库就是存放数据的地方。作为网站的开发者，你可以按照自己的意愿创建任意数量的数据库，但通常，创建一个数据库就可以满足网站的需求。这没有一定之规，但一般而言，一个网站或应用程序对应一个数据库是最合理的。

对象存储

对象存储（object store）相当于保存数据的桶。如果你使用过传统的关系型数据库，则可以将对象存储想象成一张表。大致说来，如果 Web 应用程序有一个数据库，那么，其中存储的每一种类型的数据都相应地会有一个对象存储。假如一个网站需要持久保留文献文章和用户记的笔记，那么你就可以想象它有两个对象存储。和关系型数据库表不同，对象存储不需要一个严格的列结构来指明数据如何存储。例如，在一个名为 `person` 的 MySQL 数据库表中，你可以使用两个字符类型的列存储姓和名，使用一个数值类型的列存储年龄。在 IndexedDB 中，数据存储更宽松。可以存储一个有名有姓但没有年龄（年龄未知，或年龄太大）的 `person` 对象，同时还可以另外存储一个有适当年龄的 `person` 对象。IndexedDB 让你可以更灵活地存储数据。这有利有弊，因为可以“混合”并不意味着应该混合！

索引

这是“IndexedDB”一词中“Indexed”（即索引）的由来。索引是一种从对象存储中检索数据的方式。你可以总是从对象存储中获取所有数据，但许多时候，你会希望通过一个特定的属性获取数据。例如，如果你正在存储人员信息，那么可能会希望后续通过姓名、社会保障号或者性别来获取数据。通过使用索引，你可以让 IndexedDB 系统简化通过那些属性获取数据的过程。

随着讨论进行，你会了解到其他一些重要的 IndexedDB 术语，但这三个将是你在整个开发过程中遇到的最主要的术语。

4.3　检查IndexedDB支持

因为 IndexedDB 尚未获得广泛支持，所以在实际使用之前检查浏览器对它的支持情况很重要。最简单的方法是检查 `window` 对象。

```
if("indexedDB" in window) {
}
```

当然，也可以把上述代码写成一个函数，如下所示。

```
function idbOK() {
    return "indexedDB" in window;
}
```

考虑到 iOS 8 对 IndexedDB 的支持存在严重的问题，你可能希望修改代码，以便在那些平台上返回 `false`。下面这条 StackOverflow 上的回复（http://stackoverflow.com/a/9039885）提供了一种简单实用的正则表达式文本。

```
function idbOK() {
    return "indexedDB" in window &&
    !/iPad|iPhone|iPod/.test(navigator.platform);
}
```

4.4 使用数据库

如上所述，数据库是数据的顶层容器。创建几个数据库、如何命名等都完全由你决定。创建数据库时，需要提供一个名称和版本。版本号通常从 1 开始，可以是任意值，但很重要。数据库结构（指对象存储和索引，而不是实际数据本身）只能在更改版本时调整。也就是说，如果你有一个实际的 Web 应用，并需要存储一些新类型的数据，那么就需要增加版本，生成一个新的版本号。

在 IndexedDB 中，你所做的所有操作都是异步的。因此，打开数据库并不意味着立即就可以使用，而是需要在响应一个事件之后才能开始使用。打开数据库操作可以触发的事件包括 `success`、`error`、`upgradeneeded` 和 `blocked`。

前两个就不需要解释了，但其他两个是什么意思呢？ `upgradeneeded` 在用户首次访问数据库或者版本号发生变化时被触发，这是设置数据结构的地方。`blocked` 在数据库不可用或者无法使用时被触发。示例 4-1 展示了一个打开数据库的简单场景。我们实际上没用数据库做任何事，只是尝试打开它。

示例 4-1　test_1_1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<script>
function idbOK() {
    return "indexedDB" in window;
}

var db;
```

```
$(document).ready(function() {

    //不支持？偷偷撇撇嘴
    if(!idbOK()) return;

    var openRequest = indexedDB.open("ora_idb1",1);

    openRequest.onupgradeneeded = function(e) {
        console.log("running onupgradeneeded");
    }

    openRequest.onsuccess = function(e) {
        console.log("running onsuccess");
        db = e.target.result;
    }

    openRequest.onerror = function(e) {
        console.log("onerror!");
        console.dir(e);
    }

});

</script>
</body>
</html>
```

可以看到，以上代码首先检查浏览器是否支持 IndexedDB。如果支持，则使用 `indexedDB.open` 方法打开数据库。第一个参数是数据库名称。由于一个 IndexedDB 数据库只供一个网站使用，因此不用担心该名称与其他数据库的名称冲突。第二个参数是版本号。再说一次，你可以使用任意值，但应该从 1 开始。

该方法会返回一个请求对象，你可以在上面添加事件监听器。以上代码包含除 `blocked` 之外的所有事件的监听器。第一次运行这段代码时（假设你正在使用的浏览器支持 IndexedDB），你可以在控制台中看到如图 4-1 所示的输出信息。

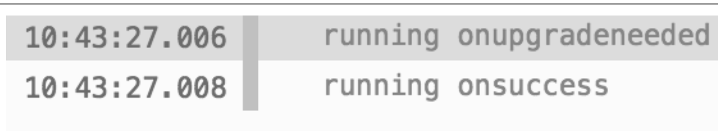


图 4-1：注意运行的事件

因为这是第一次使用数据库，所以 `upgradeneeded` 事件会被触发。同时，这也表示数据库本身创建完成。如果重复这个过程，则只有 `success` 事件会被触发（如图 4-2 所示）。



图 4-2：因为数据库已经存在，而且版本没有发生变化，所以只触发了一个事件

以上就是基本的数据库使用知识。接下来，我们将进一步讨论对象存储。

4.5　使用对象存储

前面已经说过，IndexedDB 对象存储有点像 SQL 数据库表。它应该只包含一种“类型”的数据，比如“people”“notes”或其他对象的实例。其思想是，每个需要持久化的数据类型都有一个对象存储。

对象存储只能在 upgradeneeded 事件处理期间创建。这就是版本号很重要的原因。假设你设计的数据库支持两种对象存储。数月之后，你又决定存储第三种类型的数据。你需要做两件事：第一，更改版本号；第二，编写代码，增加新的对象存储。

你可以像下面这样考虑这个过程：

```
请求打开数据库；
如果请求触发了一个upgradeneeded事件,则创建对象存储；
如果请求触发了一个success事件,则表明数据库已经准备就绪。
```

4.5.1　创建对象存储

要创建对象存储，首先应该检查它是否已经存在。可以利用数据库变量（从打开数据库操作的事件处理器获得）访问 objectStoreNames 属性。该属性是一个 DOMStringList 实例，你可以查看它是否已经包含了某个值。如果没有，则可以调用 createObjectStore("name", options) 方法创建对象存储。让我们看一下示例 4-2。

示例 4-2　test_2_1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<script>
function idbOK() {
    return "indexedDB" in window;
}
```

```
var db;

$(document).ready(function() {

    //不支持？偷偷撇撇嘴
    if(!idbOK()) return;

    var openRequest = indexedDB.open("ora_idb2",1);

    openRequest.onupgradeneeded = function(e) {
        var thisDB = e.target.result;
        console.log("running onupgradeneeded");
        if(!thisDB.objectStoreNames.contains("firstOS")) {
            console.log("makng a new object store");
            thisDB.createObjectStore("firstOS");
        }

    }

    openRequest.onsuccess = function(e) {
        console.log("running onsuccess");
        db = e.target.result;
        console.dir(db.objectStoreNames);
    }

    openRequest.onerror = function(e) {
        console.log("onerror!");
        console.dir(e);
    }

});

</script>
</body>
</html>
```

在检查并确认浏览器支持 IndexedDB 之后，打开数据库（注意，本例使用的数据库名和上个示例不同）。如果 upgradeneeded 事件被触发，则意味着用户第一次访问页面，或者数据库版本较老。

我们通过 e.target.result 取得数据库对象本身。DOMStringList 实例 objectStoreNames 让我们可以使用 contains 方法查看具有这个名称的对象存储是否已经存在。如果不存在，那就创建一个。注意，我们只传递了对象存储名称这一个参数。createObjectStore 方法还允许我们使用 options 对象传递第二个参数。使用这个参数，我们可以定义对象存储的各种配置属性，包括索引。

和以前一样，第一次运行时，upgradeneeded 事件会被触发。这次，它会做些实际的操作（如图 4-3 所示）。

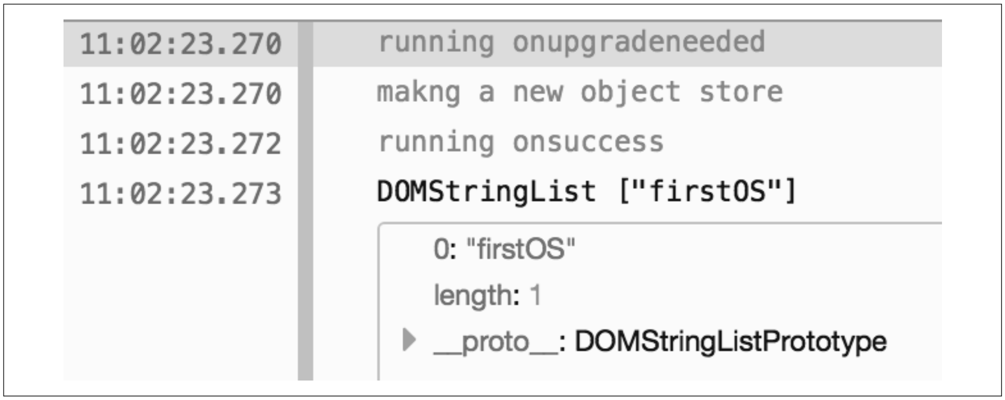


图 4-3：注意对象存储的创建

下一次请求时，只有 `success` 事件的处理器会运行，但对象存储仍然存在（如图 4-4 所示）。



图 4-4：我们的小可爱——对象存储

4.5.2　定义主键

我们即将开始讨论索引，但在开始定义不同的数据获取方式之前，你需要从一个基本属性入手，即主键。在对象存储中，每条数据都必须有一种能够唯一地标识自己的方式。例如，我的名字是 Raymond Camden，世界上当然还有其他人叫 Raymond Camden，但我的社会保障号可以唯一地标识我自己。（是的，我知道那只适用于美国人，但这不是美国人第一次表现得好像世界其他国家的人都与他们一样。）在定义对象存储时，你可以定义如何唯一地标识数据。

实际上，主要有两种定义方式。一种是定义一个 key path，它本质上是一个永远存在并且包含唯一信息的属性。因此，如果要定义对象存储 `people`，那么我会指定 key path 为 `ssn`[1]。另一种是使用 key generator，它本质上是一种生成唯一值的方式。下面举几个例子。

注 1：即 Social Security number，社会保障号。——编者注

```
somedb.createObjectStore("people", {keyPath: "email"});
```

该示例创建了一个名为 people 的对象存储，并假定每条数据都包含一个名为 email 的唯一属性。

```
somedb.createObjectStore("notes", {autoIncrement:true});
```

该示例创建了一个名为 notes 的对象存储，并使用一个自增值自动为主键赋值。

```
somedb.createObjectStore("logs", {keyPath: "id", autoIncrement:true});
```

该示例创建了一个名为 logs 的对象存储。这一次，使用自增值并将其存储在一个名为 id 的属性中。

那么，哪一个合适呢？这得视情况而定。如果存储的数据有一个属性应该唯一，那么你可能希望使用 keyPath 选项确保唯一性。如果存储的数据本身没有唯一属性，那么使用自增值就是合理的。请看示例 4-3。

示例 4-3　test_2_2.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<script>
function idbOK() {
    return "indexedDB" in window;
}

var db;

$(document).ready(function() {

    //不支持？偷偷撇撇嘴
    if(!idbOK()) return;

    var openRequest = indexedDB.open("ora_idb3",1);

    openRequest.onupgradeneeded = function(e) {
        var thisDB = e.target.result;
        console.log("running onupgradeneeded");

        if(!thisDB.objectStoreNames.contains("people")) {
            thisDB.createObjectStore("people",
                {keyPath: "email"});
        }
```

```
            if(!thisDB.objectStoreNames.contains("notes")) {
                thisDB.createObjectStore("notes",
                    {autoIncrement:true});
            }

            if(!thisDB.objectStoreNames.contains("logs")) {
                thisDB.createObjectStore("logs",
                    {keyPath:"id", autoIncrement:true});
            }

        }

        openRequest.onsuccess = function(e) {
            console.log("running onsuccess");
            db = e.target.result;
            console.dir(db.objectStoreNames);
        }

        openRequest.onerror = function(e) {
            console.log("onerror!");
            console.dir(e);
        }

    });

</script>
</body>
</html>
```

该示例的重点是 upgradeneeded 事件。它创建了三个对象存储，分别展示了不同的对象存储主键定义方式。因为我们还没有真正地存储数据，所以这里没有太多要看的东西。

4.5.3 定义索引

在指定数据的主键之后，需要确定索引。如前所述，索引定义了你计划如何从对象存储获取数据。这很大程度上取决于数据和应用程序的需求。索引必须在创建对象存储时创建，也可以用于定义数据的唯一约束（这和主键不同）。

以下代码使用对象存储变量的实例创建索引。

```
objectStore.createIndex("name of index", "path", options);
```

第一个参数是索引名称，第二个参数是你希望索引的数据属性。大多数情况下，两个参数都使用相同的值。最后一个参数是一组 options 对象，定义如何操作索引。options 对象只有两种：一种指定唯一性，另一种专门用于映射到数组的数据（稍后你会看到这样的例子）。下面是两个例子。

```
objectStore.createIndex("gender", "gender", {unique:false});
objectStore.createIndex("ssn", "ssn", {unique:true});
```

和你想的一样，第一个索引建在性别属性上，让你可以根据人的性别获取数据。第二个索引建在社会保障号属性上，并且是唯一的。

让我们看一下示例 4-4。该示例与上个示例略有不同。因此，为了突出重点，我们只列出了 `upgradeneeded` 事件处理器的代码。

示例 4-4　test_2_3.html 的一部分

```
openRequest.onupgradeneeded = function(e) {
    var thisDB = e.target.result;
    console.log("running onupgradeneeded");

    if(!thisDB.objectStoreNames.contains("people")) {
        var peopleOS = thisDB.createObjectStore("people",
            {keyPath: "email"});

        peopleOS.createIndex("gender", "gender", {unique:false});
        peopleOS.createIndex("ssn", "ssn", {unique:true});

    }

    if(!thisDB.objectStoreNames.contains("notes")) {
        var notesOS = thisDB.createObjectStore("notes",
            {autoIncrement:true});
        notesOS.createIndex("title","title", {unique:false});
    }

    if(!thisDB.objectStoreNames.contains("logs")) {
        thisDB.createObjectStore("logs",
            {keyPath:"id", autoIncrement:true});
    }

}
```

在修改后的示例中，第一个和第二个对象存储有索引。为了创建索引，我们使用了 `createObjectStore` 方法返回的实例，调用了其 `createIndex` 方法。第三个（也就是最后一个）对象存储没有索引，那完全没问题。要记住一点，每次新增、编辑或删除数据时，索引都会更新。对于 IndexedDB 而言，更多的索引意味着更大的开销。

4.6　操作数据

终于，我们讨论完了 IndexedDB 数据库的设置和初始化。我还不知道，实际地存储数据是不是令人愉快？首先，IndexedDB 的所有数据操作都是在事务中完成的。你可以将事务理解成操作的安全封装器。如果一个事务中的某个操作出现了问题，那么任何修改都会回滚。为了确保数据的一致性，事务会设置操作的安全级别。对于开发人员而言，这意味着创建、读取、更新和删除数据等简单操作（即 CRUD 操作）都要变得稍微复杂一些，与 Web 存储的简单易用相比尤其如此。IndexedDB 的事务特定于一个或多个对象存储，从根

本上说，就是你需要操作的存储。这些存储可以是只读的，也可以是可读写的。也就是说，你可以修改数据库，或者仅仅从数据库读取数据。让我们从创建数据开始介绍。

4.6.1 创建数据

要创建数据，只需调用对象存储对象的 add 方法。下面是最简单的情况。

```
someObjectStore.add(data);
```

如果对象存储要求你在创建数据时传入主键，则可以使用第二个参数，如下所示。

```
someObjectStore.add(data, somekey);
```

最酷的是，“数据”可以是任何类型的——字符串、数值、包含字符串和数值的对象，等等。和大多数操作一样，添加数据也是异步的。因此，你需要监听事件来检查添加状态。

让我们看一个例子。在开始之前，我们在浏览器中看一下这个演示程序。该演示程序有两个简单的表单：一个用于添加人员（Add Person），一个用于添加笔记（Add Note），如图 4-5 所示。

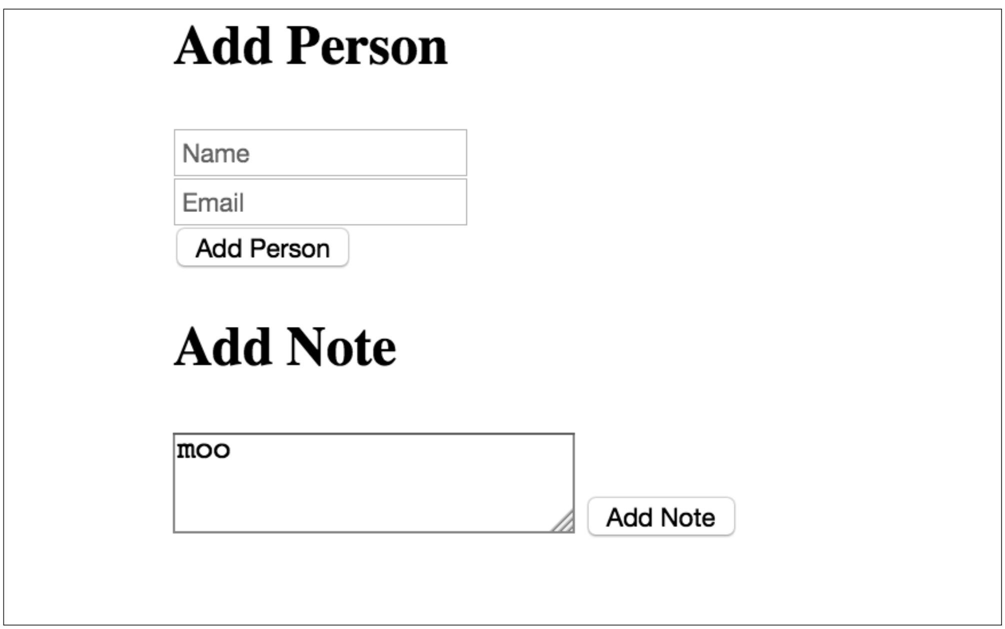


图 4-5：两个保留数据的表单

第一个表单要求输入姓名和电子邮件地址，而第二个表单要求输入字符串。该演示程序没有包含任何形式的验证，但在真实的应用程序中可以加上。在这两个表单中，你只要输入数据并点击相应的按钮，就可以从控制台看到数据输入结果（如图 4-6 所示）。

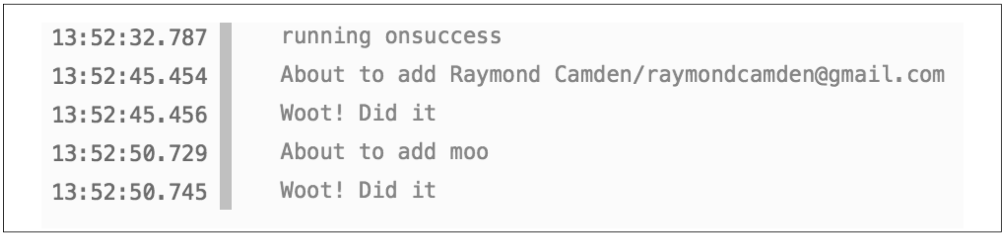


图 4-6：控制台显示数据输入成功

我们并没有实际地展示数据，但目前为止，这已经足够测试向 IndexedDB 添加数据了。现在让我们看一下代码（示例 4-5）。

示例 4-5　test_3_1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>

</head>

<body>

<h2>Add Person</h2>
<input type="text" id="name" placeholder="Name"><br/>
<input type="email" id="email" placeholder="Email"><br/>
<button id="addPerson">Add Person</button>

<h2>Add Note</h2>
<textarea id="note"></textarea>
<button id="addNote">Add Note</button>


<script>
function idbOK() {
    return "indexedDB" in window;
}

var db;

$(document).ready(function() {
    //不支持？偷偷撇撇嘴
    if(!idbOK()) return;

    var openRequest = indexedDB.open("ora_idb5",1);

    openRequest.onupgradeneeded = function(e) {
        var thisDB = e.target.result;
        console.log("running onupgradeneeded");

        if(!thisDB.objectStoreNames.contains("people")) {
```

```
            var peopleOS = thisDB.createObjectStore("people",
                {keyPath: "email"});

        }

        if(!thisDB.objectStoreNames.contains("notes")) {
            var notesOS = thisDB.createObjectStore("notes",
                {autoIncrement:true});
        }

    }

    openRequest.onsuccess = function(e) {
        console.log("running onsuccess");
        db = e.target.result;

        //开始监听按钮点击
        $("#addPerson").on("click", addPerson);
        $("#addNote").on("click", addNote);
    }

    openRequest.onerror = function(e) {
        console.log("onerror!");
        console.dir(e);
    }

});

function addPerson(e) {
    var name = $("#name").val();
    var email = $("#email").val();

    console.log("About to add "+name+"/"+email);

    //获取事务
    //默认为全部对象存储,事务类型为read
    var transaction = db.transaction(["people"],"readwrite");
    //请求objectStore
    var store = transaction.objectStore("people");

    //定义person
    var person = {
        name:name,
        email:email,
        created:new Date().getTime()
    }

    //添加数据
    var request = store.add(person);

    request.onerror = function(e) {
        console.log("Error",e.target.error.name);
        //某个类型的错误处理器
    }
```

```
        request.onsuccess = function(e) {
            console.log("Woot! Did it");
        }
    }

    function addNote(e) {
        var note = $("#note").val();

        console.log("About to add "+note);

        //获取事务
        //默认为全部对象存储,事务类型为read
        var transaction = db.transaction(["notes"],"readwrite");
        //请求objectStore
        var store = transaction.objectStore("notes");

        //定义note
        var note = {
            text:note,
            created:new Date().getTime()
        }

        //添加数据
        var request = store.add(note);

        request.onerror = function(e) {
            console.log("Error",e.target.error.name);
            //某个类型的错误处理器
        }

        request.onsuccess = function(e) {
            console.log("Woot! Did it");
        }
    }
    </script>
    </body>
    </html>
```

该演示程序的代码相当长，所以我们将一块一块地分析。首先，我们看一下 upgradeneeded 事件处理器。它定义了两个对象存储，一个名为 people，一个名为 notes。对于 people，我们将 key path email 定义为主键；对于 notes，我们使用了一个自增值。这是随意定的：对于这个演示程序，我们选用电子邮件地址作为 people 的唯一标识，而 notes 有一个人为赋予的主键。

注意，我们实际上是直到数据库 onsuccess 处理器运行，才开始处理表单提交的。这是对的，因为我们不能在数据库准备好之前添加数据。但也要注意，我们将变量 db 复制到了全局作用域。这让我们有一个数据库对象句柄，可以在后续添加数据时使用。

现在，把注意力转移到 addPerson 函数上。该函数在第一个表单提交时运行。在取得表单的值之后（再说一次，这里可以添加验证），我们开始操作 IndexedDB 数据库。首先，创建一个事务。然后，定义该事务，指定我们感兴趣的对象存储及所需的事务类型。

```
var transaction = db.transaction(["people"],"readwrite");
```

然后，我们从事务请求对象存储。

```
var store = transaction.objectStore("people");
```

现在开始有趣了。我们需要定义存储什么数据。IndexedDB 让你几乎可以存储任何你希望存储的数据。因此，存储什么数据完全取决于应用程序的需求。在本例中，我决定使用表单的值，以及一个表示人员创建时间的时间戳。说明一下，这是任意的。这里只是要说明，数据的形式由你决定。

```
var person = {
    name:name,
    email:email,
    created:new Date().getTime()
}
```

现在看一下如何保留数据。实际的存储请求相当简单：

```
var request = store.add(person);
```

但是，因为这个过程是异步的，所以我们需要监听结果。在本例中，我们监听错误事件和成功事件，并简单地使用控制台显示相关信息。`addNote` 函数的执行过程与此类似，唯一的不同是使用的对象存储和实际保存的数据。

4.6.2 读取数据

数据读取也是异步的，而且也需要使用事务。除此之外，就非常简单了：`someObjectStore.get(primaryKey)`。下面的演示程序是在上一个的基础上构建的，如图 4-7 所示。这里新增了两个表单，让你可以根据主键获取数据。

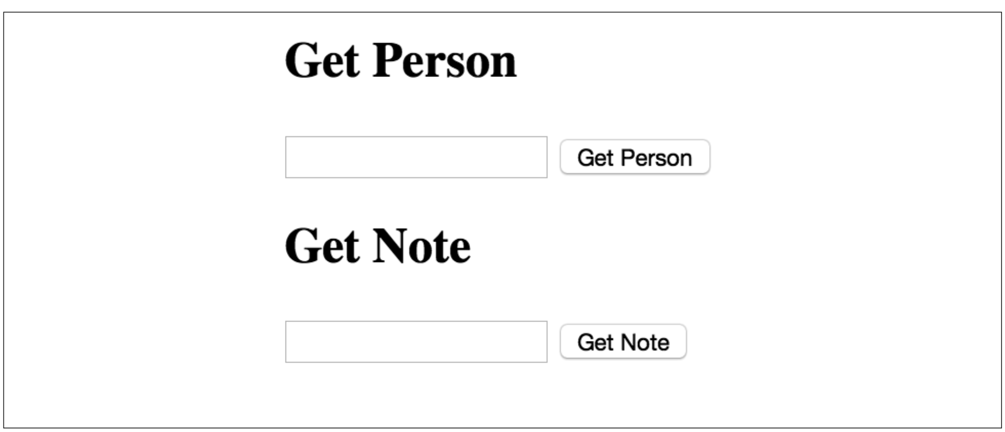


图 4-7：美观的数据检索表单

由于该演示程序的代码和上一个演示程序非常像，因此我们只需要重点看一下新表单的事件处理器（示例 4-6）。

示例 4-6　test_3_2.html 的一部分

```
function getPerson(e) {
    var key = $("#getemail").val();
    if(key === "") return;

    var transaction = db.transaction(["people"],"readonly");
    var store = transaction.objectStore("people");

    var request = store.get(key);

    request.onsuccess = function(e) {
        var result = e.target.result;
        console.dir(result);
    }

    request.onerror = function(e) {
        console.log("Error");
        console.dir(e);
    }

}

function getNote(e) {
    var key = $("#getnote").val();
    if(key === "") return;

    var transaction = db.transaction(["notes"],"readonly");
    var store = transaction.objectStore("notes");

    var request = store.get(Number(key));

    request.onsuccess = function(e) {
        var result = e.target.result;
        console.dir(result);
    }

    request.onerror = function(e) {
        console.log("Error");
        console.dir(e);
    }

}
```

先看一下 `getPerson` 函数。在取得希望加载的主键的值之后，我们再次创建了一个事务。注意，这次创建的是一个只读事务。然后，我们只需要像下面这样获取数据。

```
var request = store.get(key);
```

在 `success` 事件处理器中，我们将结果显示在控制台上，如图 4-8 所示。

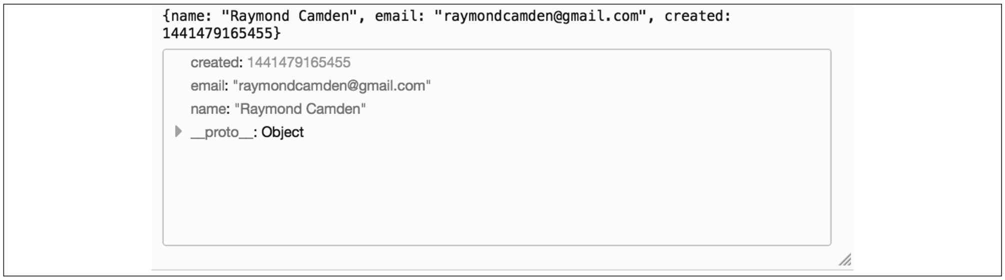


图 4-8：一个数据库对象的数据

如果你试图获取一个不存在的对象，那么 success 事件处理器仍然会运行，但结果未定义。为了让这个演示程序正常运行，你至少要创建一条记录，并记下使用过的电子邮件地址。（在本章后面，我们将看一下如何使用开发者工具查看 IndexedDB，看看它存储了什么数据。）

4.6.3　更新数据

可能你已经猜到我要讲什么了。你需要再次获取一个事务，然后使用事务返回的对象存储变量，调用其 put 方法存储数据。这可以像 someObjectStore.put(data) 一样简单，但你也可以使用第二个参数指定主键。

下面的演示程序要稍微复杂一些。现在，你需要输入现有人员的电子邮件地址（再次提醒一下，务必输入与上个演示程序完全相同的数据）。当你输入的电子邮件地址所对应的人员存在时，程序会填写表单。这样，你就可以更新数据了（如图 4-9 所示）。

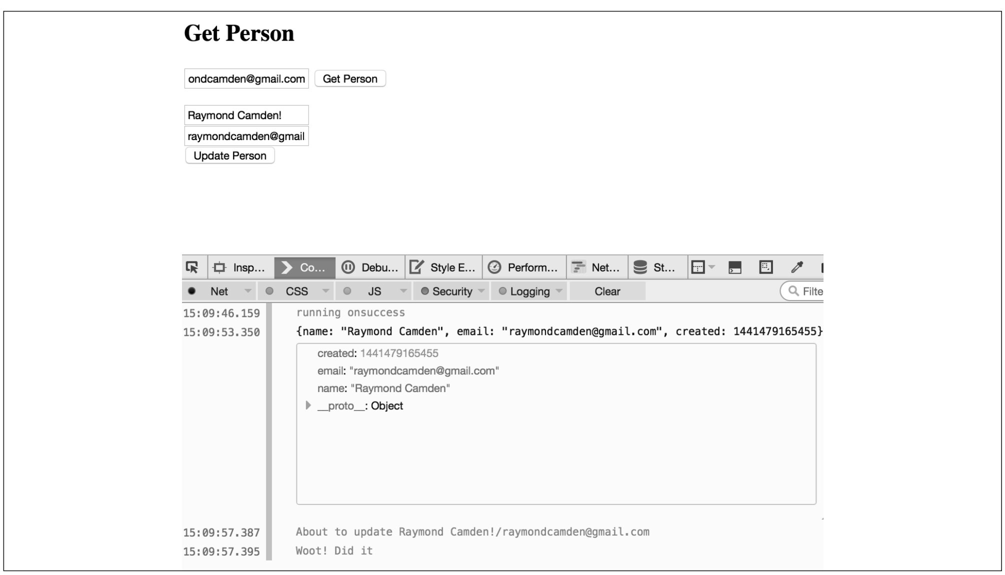


图 4-9：数据更新示例

加载人员信息的代码和上一个演示程序一样。示例 4-7 展示了这个例子的关键代码。

示例 4-7 test_3_3.html 的一部分

```
function getPerson(e) {
    var key = $("#getemail").val();
    if(key === "") return;

    var transaction = db.transaction(["people"],"readonly");
    var store = transaction.objectStore("people");

    var request = store.get(key);

    request.onsuccess = function(e) {
        var result = e.target.result;
        console.dir(result);
        $("#name").val(result.name);
        $("#email").val(result.email);
        $("#created").val(result.created);
    }

    request.onerror = function(e) {
        console.log("Error");
        console.dir(e);
    }

}

function updatePerson(e) {
    var name = $("#name").val();
    var email = $("#email").val();
    var created = $("#created").val();

    console.log("About to update "+name+"/"+email);

    //获取事务
    //默认为全部对象存储,事务类型为read
    var transaction = db.transaction(["people"],"readwrite");
    //请求objectStore
    var store = transaction.objectStore("people");

    var person = {
        name:name,
        email:email,
        created:created
    }

    //执行更新
    var request = store.put(person);

    request.onerror = function(e) {
        console.log("Error",e.target.error.name);
        //某种类型的错误处理器
    }
```

```
        request.onsuccess = function(e) {
            console.log("Woot! Did it");
        }
    }
```

getPerson 函数的代码和上一个例子类似。现在，我们使用该方法的返回结果做一些具体的操作：更新表单。updatePerson 函数只需要获取表单的值，并使用刚刚介绍的 put 方法将其持久化。重申一下，为了让应用程序更稳定，上述代码中有多处可以添加验证，但这里，你知道就行了。

4.6.4　删除数据

现在，让我们看一下 CRUD 操作的最后一部分——删除数据。同样，该操作也要在事务中进行，而且也是异步的。删除方法很简单：someObjectStore.delete(primaryKey)。同样，最后一个演示程序也很简单。它会提示你输入一个人的电子邮件地址，然后删除那个人的相关信息（如图 4-10 所示）。



图 4-10：删除人员（听起来很残酷）

示例 4-8 展示了点击 Delete Person 按钮后执行的代码。

示例 4-8　test_3_4.html 的一部分

```
function deletePerson(e) {
    var key = $("#email").val();
    if(key === "") return;

    var transaction = db.transaction(["people"],"readwrite");
    var store = transaction.objectStore("people");

    var request = store.delete(key);

    request.onsuccess = function(e) {
        console.log("Person deleted");
        console.dir(e);
    }

    request.onerror = function(e) {
        console.log("Error");
        console.dir(e);
    }

}
```

注意，即使要删除的人不存在，删除操作也会触发 `success` 事件处理器。如果你想把这种情况当成错误来处理，则需要首先获取那个人的信息，看一下返回结果是否已定义，然后再执行删除操作。为了确保数据一致，整个过程都由事务控制，这和传统的关系型数据库中的事务非常像。

4.7　获取所有数据

现在，你已经了解了基本的 CRUD 操作。下面，我们将讨论如何获取数据库中的所有（和部分）数据。IndexedDB 使用一个名为*游标*（cursor）的东西遍历对象存储中的数据。你可以将游标想象成一只快乐的小海狸，它跑进对象存储，一次取回一条数据。它每取得一条数据就带回来给你，而你又要求它去取下一条。游标可以双向移动（海狸也可以），也可以被限制在一定的数据“范围”内（这不太适合海狸，它们是自由的精灵）。

和 CRUD 操作一样，游标也是在事务内使用。和以前一样，你要获取一个事务，从这个事务中获取一个对象存储，然后在此基础上打开一个游标。下面是一个抽象的例子。

```
var transaction = db.transaction(["test"], "readonly");
var objectStore = transaction.objectStore("test");
var cursor = objectStore.openCursor();

cursor.onsuccess = function(e) {
    var res = e.target.result;
    if(res) {
```

```
            //操作
            res.continue();
        }
    }
```

注意游标的 success 事件处理器。事件结果包含“海狸游标”当前持有的数据。它还有一个 continue 方法。你可以使用该方法，告诉游标获取下一个对象。如果返回结果未定义，那就表示已经到达了游标末尾。

如图 4-11 所示，新演示程序现在包含了一个列出数据库中所有人的方法（以及一个添加方法，以防你在上一个例子中把所有人都删除了）。

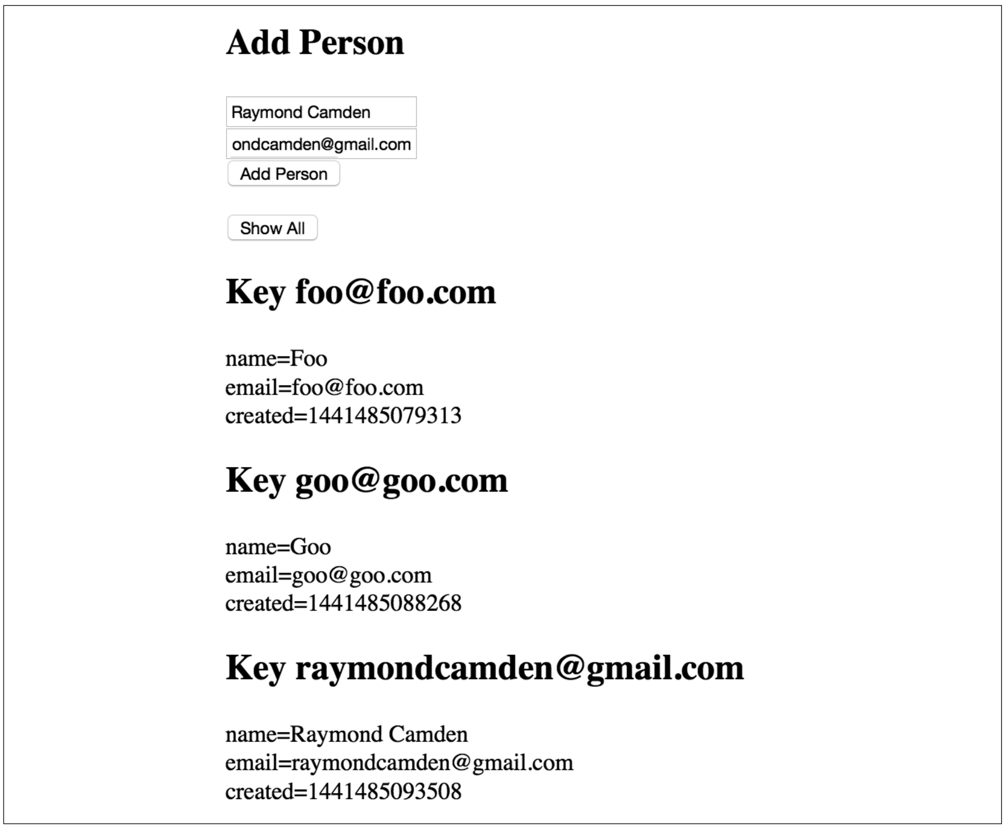


图 4-11：列出数据的例子

因为前面已经介绍过添加操作的代码，所以这里我们只关注列出数据的代码（示例 4-9）。

示例 4-9　test_4_1.html 的一部分

```
    function getPeople(e) {

        var s = "";
```

```
        var transaction = db.transaction(["people"], "readonly");
        var people = transaction.objectStore("people");
        var cursor = people.openCursor();

        cursor.onsuccess = function(e) {
            var cursor = e.target.result;
            if(cursor) {
                s += "<h2>Key "+cursor.key+"</h2><p>";
                for(var field in cursor.value) {
                    s+= field+"="+cursor.value[field]+"<br/>";
                }
                s+="</p>";
                cursor.continue();
            }
        }

        transaction.oncomplete = function() {
            $("#results").html(s);
        }
    }
```

不出所料，首先需要获取一个事务，然后获取一个存储，最后打开游标。每次取得一个对象，`success` 事件处理器就会运行。为了更新 Web 页面，我们将使用变量 `s` 存放表示数据的 HTML。注意，这里使用类似 Handlebars（http://www.handlebarsjs.com）这样的模板语言会更好一些。游标对象包含一个代表数据项主键的键属性，还包含一个代表数据的值属性。我们可以遍历对象中的每个键，并追加到字符串。在一个“真实”的应用程序中，你不会这样做。你会知道数据包含的属性，并直接把它们输出。以上只是一段简单普通的代码。

最后一部分很关键。如何知道游标遍历是否已经完成？当无法再取到数据时，事务对象会触发一个 `complete` 事件。我们可以使用它取得字符串变量，并注入 DOM。

使用范围和索引

前面介绍的游标的例子适用于显示所有数据，但通常，你会希望只操作数据的一个子集。这就是索引的用途所在了。索引以数据的一个属性为基础。你可以从那些数据中请求位于一定范围内的数据。

假设对象存储 `people` 有一个建立在 `name` 上的索引。你可以请求名字首字母大于等于 B（比如 B、C、D 等）的范围内的数据，也可以请求名字首字母介于“最小”值和 T 之间的数据。最后，你还可以请求首字母介于 R 和 S 之间的数据。

若要更复杂，对于前面所有的例子，你可以在开闭模式之间切换。这是什么意思呢？假设有一个范围介于 B 和 E 之间。闭区间包含 B 和 E 本身，可以提供以 B 和 E 开头的名字，比如 Barry 和 Elric。开区间提供 B 和 E 之间的值，但不包含以 B 和 E 开头的名字，这样请求名字的第一个返回结果可能是 Corwin。（没错，数值范围也是可以的。）

最后，你还可以创建只包含一个值的“范围”，比如以 R 开头的名字（如 Raymond）。

使用范围和使用游标稍有不同。范围是在索引上打开游标，而不是在对象存储上，如下所示。

```
//创建一个IDBKeyRange
range = IDBKeyRange.upperBound("Camden");
cursor = someIndex.openCursor(range);
//或者
cursor = someIndex.openCursor(range, "prev");
```

注意，以上代码中的范围设置了上限 `"Camden"`。也就是说，在字符串比较时，名字要“低于”Camden，比如 Cameron 不低于 Camden，而 Cade 就低于 Camden。

范围是由 IDBKeyRange API 创建的。该 API 提供了 `upperBound`、`lowerBound`、`bound`（包含上限和下限）和 `only` 函数。范围会自动设置为闭区间，但你可以把 `false` 传递给第二个（或者 `bound` 函数的第三个）参数，将其设置为开区间。在默认情况下，游标方向是 `"forward"`，但在最后一个例子中，你可以看到如何指定向后遍历。

所有这些都相当复杂，让我们看一个例子。该演示程序经过了扩展，现在你可以通过名字搜索人员了，如图 4-12 所示。你可以指定搜索名字以某个字母开头，或以某个字母结尾，或首字母介于某两个字母之间的人。

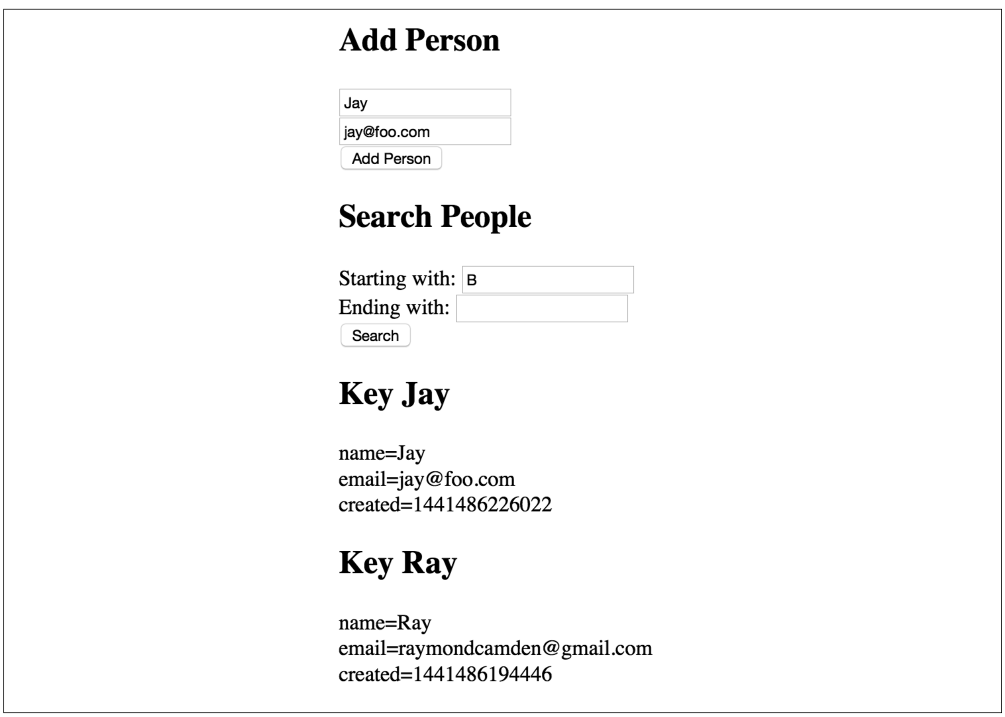


图 4-12：人员搜索表单

在尝试运行这段代码之前，应该注意一点，就是该示例使用了一个新的 IndexedDB 数据库。务必在最上面的 Add Person 表单里输入一些值，以便你有数据可以进行实际的检索。由于添加人员不是新功能，因此我们只关注检索部分的代码，如示例 4-10 所示。

示例 4-10　test_4_2.html 的一部分

```
function searchPeople(e) {

    var lower = $("#lower").val();
    var upper = $("#upper").val();

    if(lower == "" && upper == "") return;

    var range;
    if(lower != "" && upper != "") {
        range = IDBKeyRange.bound(lower, upper);
    } else if(lower == "") {
        range = IDBKeyRange.upperBound(upper);
    } else {
        range = IDBKeyRange.lowerBound(lower);
    }

    var transaction = db.transaction(["people"],"readonly");
    var store = transaction.objectStore("people");
    var index = store.index("name");

    var s = "";

    index.openCursor(range).onsuccess = function(e) {
        var cursor = e.target.result;
        if(cursor) {
            s += "<h2>Key "+cursor.key+"</h2><p>";
            for(var field in cursor.value) {
                s+= field+"="+cursor.value[field]+"<br/>";
            }
            s+="</p>";
            cursor.continue();
        }
    }

    transaction.oncomplete = function() {
        //没有结果?
        if(s === "") s = "<p>No results.</p>";
        $("#results").html(s);
    }


}
```

检索函数首先读取了检索表单的值，并做了一点验证。在这时，事情就变得有点微妙了。还记得吗？我们可以从某个字母开始检索，也可以检索到某个字母，还可以在两个字母之间检索。也就是说，我们需要三种范围类型中的一种。这就是接下来的代码块所做的工

作。它会根据输入确定使用哪一种范围类型合适。这一步完成之后，接下来就是打开事务，获取对象存储，然后检索名字索引。

在获取游标时，范围会作为一个参数传递。除此之外，游标对象的处理和示例 4-9 一样。

你可能想知道更复杂的检索如何实现——例如，检索已知名字首字母和性别、年龄在 10 到 30 之间的人。遗憾的是，复杂的检索不是 IndexedDB 擅长的事情。它不会取代像 MySQL 这样的真正的 SQL 数据库引擎。关于这一点，在构建应用程序时要牢记在心。

4.8 关于IndexedDB的更多内容

关于 IndexedDB，我们还有一些细节没有介绍。让我们再看两个有趣的使用技巧。

4.8.1 存储数组

上文已经提到，IndexedDB 可以存储几乎任何数据，甚至是数组数据。例如：

```
var person = {
    name:"Ray",
    age:43,
    background:{
        born:1973,
        bornIn:"Virginia"
    },
    hobbies:["comics","movies","bike riding"]
}
someStore.add(person);
```

没错，这样的确很棒，而且只是一项很简单的工作，但它提出了一个有趣的问题：如果希望根据业余爱好获取人员信息，那该怎么做？好，这就是 `multiEntry` 选项的用途所在了。当在一个基于数组的属性上定义索引时，只需使用这个选项并设置其为 `true`。

```
objectStore.createIndex("hobbies", "hobbies", {unique:false, multiEntry:true});
```

该语句告诉 IndexedDB，将数组中的每个数据项以恰当的方式存储到索引中，这样你就可以根据某个特定的值获取某个人的信息。让我们看一下演示程序，如图 4-13 所示。

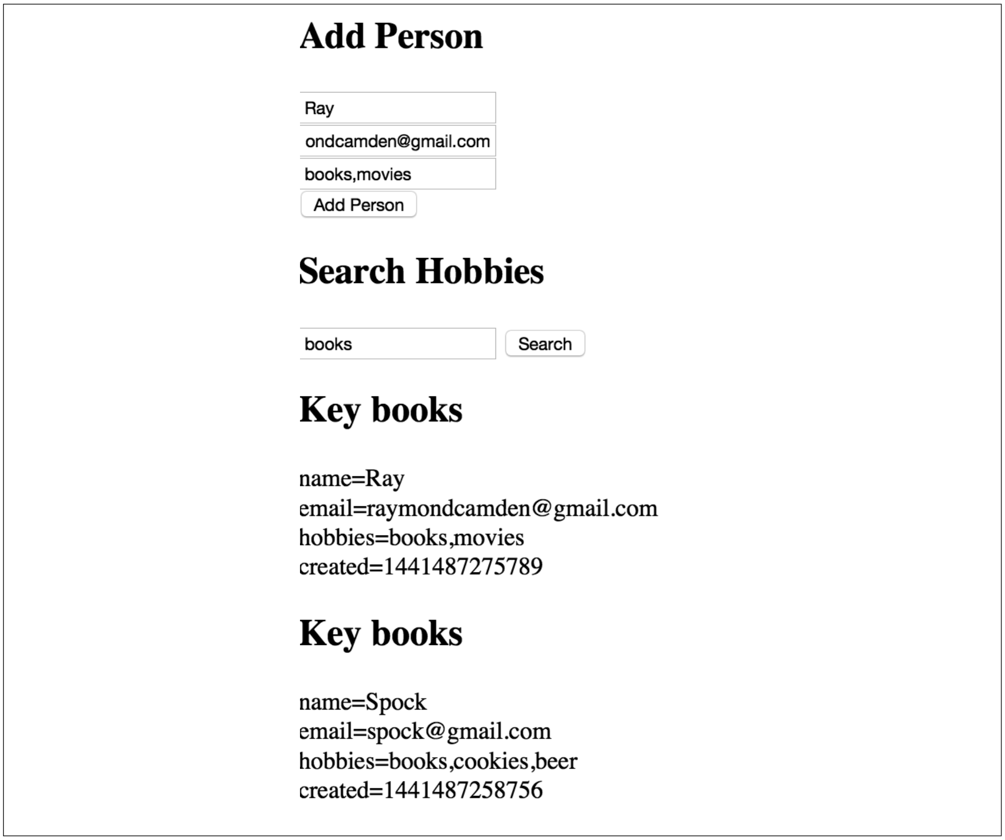


图 4-13：人员信息现在包含业余爱好

从图 4-13 中可以看到，更新后的 Add Person 表单包含了一个业余爱好字段。在测试的时候，应该输入用逗号分隔的爱好列表，而且爱好之间不能有空格，例如 `cookies,beer,movies`，而不可以是 `cookies, beer, movies`。（重申一下，在正式发布的应用程序中，你可以在代码中删除空格。）现在可以基于爱好检索了。输入一种爱好的名称，你就可以找出所有有此爱好的人。让我们看一下代码，由于它和前面的例子有点不一样，因此我们提供了完整的代码清单（示例 4-11）。

示例 4-11　test_5_1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src=
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>
```

```
<h2>Add Person</h2>
<input type="text" id="name" placeholder="Name"><br/>
<input type="email" id="email" placeholder="Email"><br/>
<input type="text" id="hobbies" placeholder="Hobbies"><br/>
<button id="addPerson">Add Person</button>
<p/>

<h2>Search Hobbies</h2>
<input type="text" id="hobby"><button id="search">Search</button>

<div id="results"></div>

<script>
function idbOK() {
    return "indexedDB" in window;
}

var db;

$(document).ready(function() {

    //不支持？偷偷撇撇嘴
    if(!idbOK()) return;

    var openRequest = indexedDB.open("ora_idb7",1);

    openRequest.onupgradeneeded = function(e) {
        var thisDB = e.target.result;
        console.log("running onupgradeneeded");

        if(!thisDB.objectStoreNames.contains("people")) {
            var peopleOS = thisDB.createObjectStore("people",
                {keyPath: "email"});

                peopleOS.createIndex("name", "name",
                {unique:false});
                peopleOS.createIndex("hobbies", "hobbies",
                {unique:false, multiEntry: true});

        }

    }

    openRequest.onsuccess = function(e) {
        console.log("running onsuccess");
        db = e.target.result;

        //开始监听按钮点击
        $("#addPerson").on("click", addPerson);
       $("#search").on("click", searchPeople);
    }

    openRequest.onerror = function(e) {
        console.log("onerror!");
        console.dir(e);
```

```
    }

});

function addPerson(e) {
    var name = $("#name").val();
    var email = $("#email").val();
    var hobbies = $("#hobbies").val();

    if(hobbies != "") hobbies = hobbies.split(",");

    console.log("About to add "+name+"/"+email);

    //获取事务
    //默认为全部对象存储,事务类型为read
    var transaction = db.transaction(["people"],"readwrite");
    //请求objectStore
    var store = transaction.objectStore("people");

    //定义person
    var person = {
        name:name,
        email:email,
        hobbies:hobbies,
        created:new Date().getTime()
    }

    //添加数据
    var request = store.add(person);

    request.onerror = function(e) {
        console.log("Error",e.target.error.name);
        //某个类型的错误处理器
    }

    request.onsuccess = function(e) {
        console.log("Woot! Did it");
    }
}

function searchPeople(e) {

    var hobby = $("#hobby").val();

    if(hobby == "") return;

    var range = IDBKeyRange.only(hobby);

    var transaction = db.transaction(["people"],"readonly");
    var store = transaction.objectStore("people");
    var index = store.index("hobbies");

    var s = "";

    index.openCursor(range).onsuccess = function(e) {
```

```
        var cursor = e.target.result;
        if(cursor) {
            s += "<h2>Key "+cursor.key+"</h2><p>";
            for(var field in cursor.value) {
                s+= field+"="+cursor.value[field]+"<br/>";
            }
            s+="</p>";
            cursor.continue();
        }
    }

    transaction.oncomplete = function() {
        //没有结果?
        if(s === "") s = "<p>No results.</p>";
        $("#results").html(s);
    }

}

</script>
</body>
</html>
```

这段代码相当长，但实际上变化很少。首先，请注意关于人员的新索引：

```
peopleOS.createIndex("hobbies", "hobbies",
{unique:false, multiEntry: true});
```

如前所述，设置为 true 的 multiEntry 选项是这一切可以正常运行的魔法标识。现在，向下滚动到 addPerson 函数的逻辑。为了存储数组，我们只需要将表单中的字符串值转换成一个 JavaScript 数组：

```
if(hobbies != "") hobbies = hobbies.split(",");
```

最后，搜索需要找出完全匹配的数据项，因此，使用 only 方法取代指定了起点和终点的范围。

```
var range = IDBKeyRange.only(hobby);
```

4.8.2 计算数据量

本章最后一个演示程序将展示如何计算对象存储中的数据量。你可能认为需要使用游标遍历整个表。不过，有一种简单许多的数据量计算方法：使用 count 方法。对象存储的 count 方法，其用途和你想的完全一样——异步返回存储中对象的数量。下面是一个例子。

```
db.transaction(["note"],"readonly").objectStore("note").count().onsuccess =
function(event) {
    console.log('total is '+event.target.result);
}
```

请注意，我们在一行代码中将各种方法调用灵活地链接在了一起，同事会认为我们很酷，但那完全没有必要。实际的 `count` 值包含在事件结果值中。本书提供的代码中有一个这样的例子（test_5_2.html）。

4.9　使用开发者工具查看IndexedDB

对于 Web 存储，Firefox 和 Chrome 都提供了优秀的工具，让你可以操作 IndexedDB。如图 4-14 所示，这是 Firefox 对 IndexedDB 的支持。

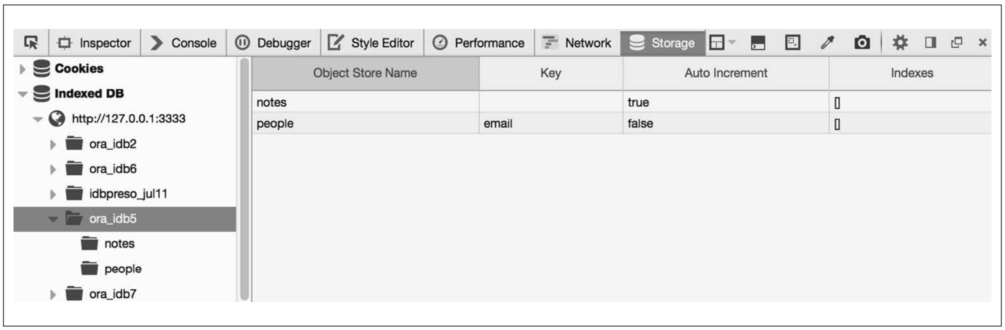


图 4-14：Firefox 开发者工具对 IndexedDB 的支持

除了可以查看数据库和对象存储的高级视图外，你还可以选择一个存储，查看详细的值列表（如图 4-15 所示）。

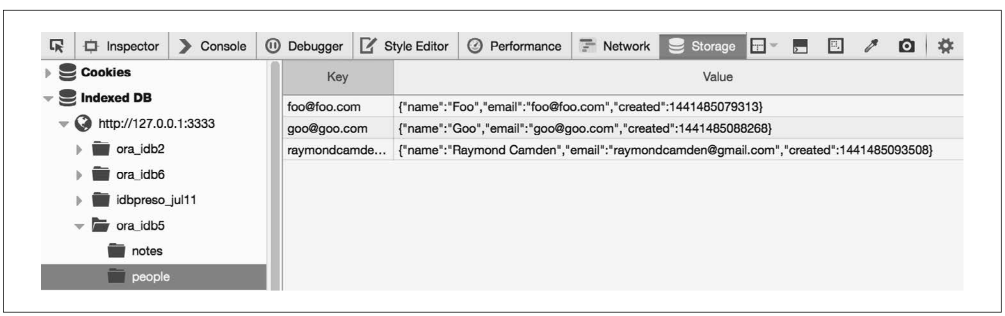


图 4-15：数据视图

图 4-16 展示了 Chrome 如何显示对象存储。注意底部带斜杠的圆圈。你可以用它快速删除数据。

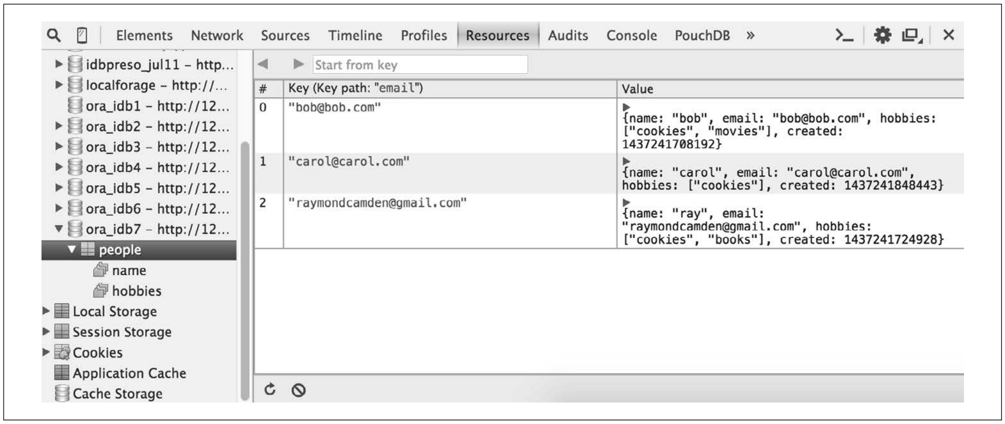


图 4-16：Chrome 中的 IndexedDB 视图

4.10 浏览器支持和使用建议

那么，IndexedDB 的浏览器支持情况如何呢？图 4-17 是来自 CanIUse.com 的浏览器支持报告。

邮
电

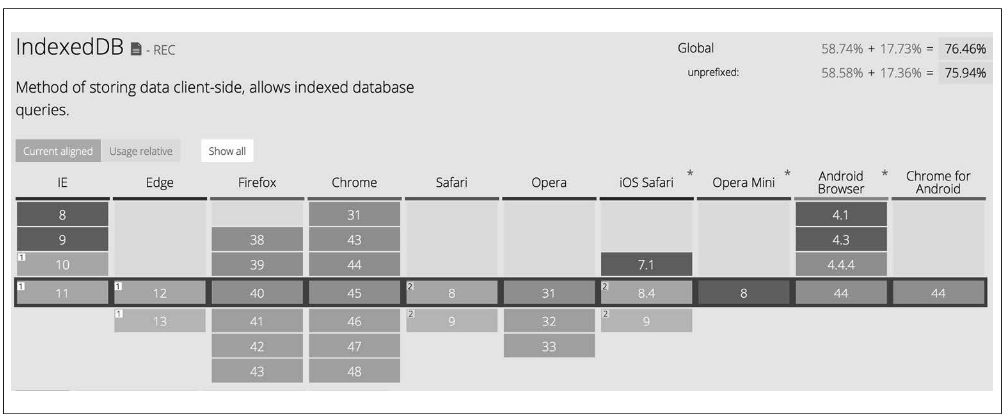


图 4-17：来自 CanIUse.com 的 IndexedDB 浏览器支持报告

IndexedDB 的浏览器支持情况还算不错，但并不是特别好。不过，支持情况正变得越来越好，甚至 iOS 很快都（可能）要提供支持了。

至于建议用法，我认为，用户能够创建的任何内容都可以使用 IndexedDB 存储。你可以用它将非私有的内网信息复制到本地，以便获得更快的检索速度，并实现离线支持。你也可以复制游戏资源，比如小型音乐文件和游戏数据。

第5章

使用Web SQL

5.1 已废弃的规范

在开始讨论 Web SQL 之前，我很遗憾地告诉你，这份规范已经废弃，或者说即将废弃，或者说至少是必将废弃。Web SQL 是一个颇为有趣的特性。它使得你可以在浏览器中访问微型数据库。对于从事服务器端开发的 Web 开发人员而言，这特别有吸引力，因为他们可能对 SQL 已经有了一定的了解。然而，由于一些对于本书而言并不重要的原因，这份规范已经走到了生命的尽头，未来（可能）将不复存在。理论上，这意味着你甚至不应该读这一章。

然而……

Web SQL 在移动端浏览器中获得了很好的支持，它先于 IndexedDB 而存在，而且比 IndexedDB 获得了更好的支持。作为开发人员，你完全有可能遇到使用了 Web SQL 的 Web 应用。我虽然不建议在新项目中使用 Web SQL，但希望本章能够让你对 Web SQL 有充分的了解，以便在不得不为现有的实现提供支持和帮助时，知道怎么做。

和前面讨论的技术一样，该客户端数据存储技术也与域名一一对应。存储限制差别很大，从 5MB 到 50MB，甚至更大。在深入介绍这项技术之前，让我们先了解几个基本术语。

5.2 数据库基本术语

如果你已具备传统关系型数据库服务器的使用经验，那么可以跳过这一节，继续往下阅读。

数据库

数据库是存储数据的顶层容器。与 IndexedDB 一样，可创建的数据库数量没有限制，但通常来说，一个网站几乎总是依附于一个数据库。

表

表用于存储一种特定类型的数据。与 IndexedDB 的对象存储不同，表对存储的内容有非常严格的要求。如果你定义了一个存储人员信息的表，它包含姓名、年龄和性别三列，那么就只能将值存储在这些列中。每一列都预设了特定的数据类型，这是数据存储必须满足的条件。

行

行是表中独立的数据单元。在上述的人员信息表中，一行数据就代表一个人。

5.3 检查Web SQL支持

要检查 Web SQL 支持，最简单的方法是检查 `window` 对象的 `openDatabase` API，如下所示。

```
if("openDatabase" in window) {
}
```

为了友好起见，可以将以上语句换成类似下面这样的函数。

```
function webSQLOK() {
    return "openDatabase" in window;
}
```

5.4 使用数据库

与 IndexedDB 非常类似，Web SQL 数据库也有一个名称和版本号。不过，与 IndexedDB 不同的是，版本号不会触发一个事件供你执行更新。相反，它会承担验证功能。如果用户有一个数据库的早期版本，那么你可以通过手动执行更新来处理变化。（不过，我们会展示一种更简单的方法。）接下来，为数据库提供一个“友好的名称”。据我所知，这个名称不会再用到。你还需要指定数据库的初始大小。这是一个估计值，老实说，我见过的大多数示例都使用了同一个值（5MB），而且开发人员实际上并没有花心思去弄清楚自己真正需要多大的数据库。你的代码将从打开数据库开始，这是一个同步 API。

```
db = window.openDatabase("name","1","nice name",5*1024*1024);
```

请注意，我取得了 `window.openDatabase` 调用的返回结果，并将其存储到了变量 `db` 中。稍后，你可以用它在数据库中执行操作。让我们考虑一个简单的例子：打开数据库，并通过控制台显示数据库对象本身（示例 5-1）。

示例 5-1　test1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<script>
function websqlOK() {
    return "openDatabase" in window;
}

var db;

$(document).ready(function() {

    //不支持？偷偷撇撇嘴
    if(!websqlOK()) return;

    db = window.openDatabase("db1", "1", "Database 1", 5*1024*1024);

    console.dir(db);

});

</script>
</body>
</html>
```

虽然这里没有太多需要介绍的内容，但该示例展示了使用 Web SQL 的基本代码。图 5-1 展示了 Chrome 控制台显示的数据库对象。

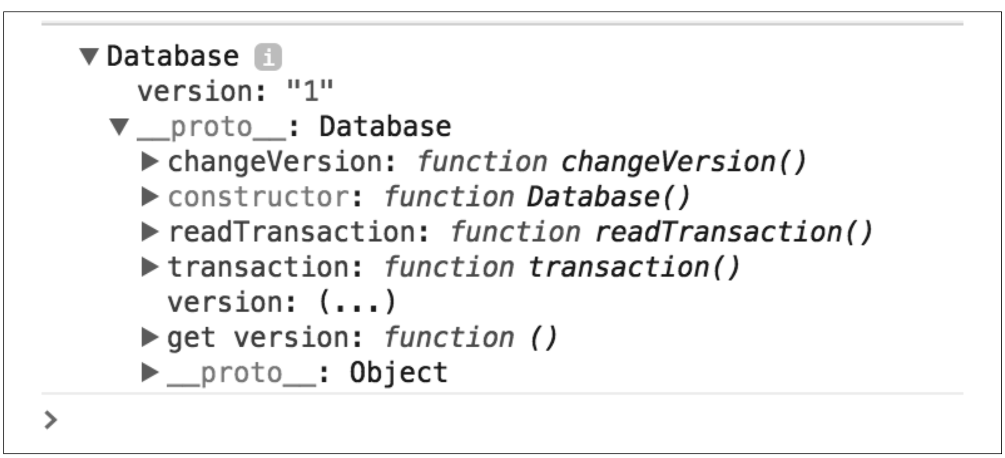


图 5-1：Chrome 转储的数据库对象

5.5 使用事务

在取得数据库对象之后，就可以开始进行各种操作了。与 IndexedDB 不同，操作 Web SQL 中的数据相当简单（当然，前提是你了解 SQL）。你将再次用到事务，并且需要将事务指定为只读或可读 / 写。不过，不管进行什么操作，接下来的代码就都一样了，唯一的变化是 SQL。下面这个基本的例子将演示如何打开一个只读事务（前提是你已经创建了一个 Web SQL 变量 db）。

```
db.readTransaction(function to do stuff, error handler, success handler);
```

实际的代码可能如下所示。

```
db.readTransaction(function(tx) {
    tx.executeSql("select * from foo");
}, function(e) {
    console.log("Db error ",e);
}, function() {
    console.log("Done");
});
```

readTransaction 函数的第一个参数是一个以事务对象为参数的函数。你可以在那个对象上运行 executeSql 函数。可能你已经猜到，这就是执行 SQL 查询的地方。第二个参数是错误处理器，最后一个参数是 success 事件处理器。

目前为止，一切顺利。但这也是情况变得让人有点困惑的地方。首先看看该 API 的基本语法。

```
tx.executeSql("sql statement", "array of values", "success handler",
"error handler");
```

现在先不管第二个参数，我们还会回过头来看它。你最需要注意的是，处理器的顺序（先是 success 事件处理器，然后是错误处理器）和事务调用中处理器的顺序相反。这很容易混淆，所以，使用这些处理器时要格外小心。

让我们扩展一下最初的演示程序，做一些设置工作。在数据库中存储数据之前，你需要有一张表。所幸，在 SQL 中创建表并不困难。示例 5-2 演示了如何创建一张表。

示例 5-2 test2.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<script>
```

```
function websqlOK() {
    return "openDatabase" in window;
}

var db;

$(document).ready(function() {

    //不支持？偷偷撇撇嘴
    if(!websqlOK()) return;

    db = window.openDatabase("db1", "1", "Database 1", 5*1024*1024);

    db.transaction(function(tx) {
        tx.executeSql("create table if not exists notes(id INTEGER PRIMARY "+
        "KEY AUTOINCREMENT, title TEXT, body TEXT, updated DATE)");
    },dbError,function(tx) {
        ready();
    });

});

function dbError(e) {
    console.log("Error", e);
}

function ready() {
    onsole.log("Ready to do stuff!");
}
</script>
</body>
</html>
```

在这个版本中，数据库打开之后，我们创建了一个读 / 写事务（使用 db.transaction）。然后，我们使用 SQL 创建了一张表。这段 SQL 已经完美地处理了表已经存在的情况。在这种情况下，它不做任何操作，这是一件很酷的事情。上文已经提到过，Web SQL 确实包含版本的概念，也确实提供了一种在版本变化时执行任务的方法，但这种设置形式要简单许多，而且多半能满足你的需求。你可以在那里执行多段不同的 SQL 语句，根据需要创建任意数量的表。

显然，如果你了解 SQL，则所有这些都很简单。如果你不了解，则有许多书籍和教程可以帮你了解。示例 5-2 使用 SQL 创建了一张名为 notes 的表。它有一个存储数据主键的列 id，两个存储文本的列 title 和 body，以及一个存储日期值的列 updated。

现在，让我们更进一步，看一下示例 5-3 这个真实而简单的演示程序。

示例 5-3　test3.html

```
<!doctype html>
<html>
<head>
```

```
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<h2>Add a Note</h2>
<form>
Title: <input type="text" id="title"><br/>
Body:<br/>
<textarea id="body"></textarea><br/>
<button id="addNote">Add Note</button>
</form>

<p/>

<table id="notes" border="1"><tbody></tbody></table>

<script>
function websqlOK() {
    return "openDatabase" in window;
}

var db;

$(document).ready(function() {

    //不支持？偷偷撇撇嘴
    if(!websqlOK()) return;

    db = window.openDatabase("db1", "1", "Database 1", 5*1024*1024);

    db.transaction(function(tx) {
        tx.executeSql("create table if not exists notes(id INTEGER PRIMARY "+
        "KEY AUTOINCREMENT, title TEXT, body TEXT, updated DATE)");
    },dbError,function(tx) {
        ready();
    });

});

function dbError(e) {
    console.log("Error", e);
}

var $title, $body, $notesTable;

function ready() {
    $("#addNote").on("click", addNote);
    $title = $("#title");
    $body = $("#body");
    $notesTable = $("#notes tbody");
    renderNotes();
}
```

```
function addNote(e) {
    e.preventDefault();
    //没有验证
    var title = $title.val();
    var body = $body.val();

    db.transaction(function(tx) {
        tx.executeSql("insert into notes(title,body,updated) values(" +
        "'" + title + "','" + body + "'," + (new Date().getTime()) +")");
    },dbError,function(tx) {
        $title.val("");
        $body.val("");
        renderNotes();
    });

}

function renderNotes() {
    db.readTransaction(function(tx) {
        tx.executeSql("select * from notes order by updated desc",[],
        function(tx, results) {
            var rowStr = "";
            for(var i=0;i<results.rows.length;i++) {
                var row = results.rows.item(i);
                //使用row.col
                rowStr += "<tr><td>" + row.title + "</td>";
                rowStr += "<td>" + row.body + "</td>";
                var d = new Date();
                d.setTime(row.updated);
                rowStr += "<td>" + d.toDateString() + " " + d.toTimeString();
                rowStr += "</td></tr>";
            };
            $notesTable.empty();
            $notesTable.append(rowStr);
        });
    },dbError);

}
</script>
</body>
</html>
```

这个新版本的演示程序包含一个表单和一个空表格。表单供用户输入笔记（标题和正文），而表格用于展示已有的数据。用户界面就这些内容，现在让我们研究一下代码。

`ready` 函数会在开头部分创建表的 SQL 执行完成之后运行。记住，这段 SQL 会运行多次，但这没什么问题，因为它不会重新创建第一次运行时创建的表。我们在表单上添加一个简单的点击事件处理器，存储几个从 DOM 取值的 jQuery 变量，并立即运行 `renderNotes` 函数。

点击事件处理器 `addNote` 只获取表单值，然后创建一条 SQL 语句处理插入。该 SQL 语句非常容易出问题。我们会在演示程序的下一个修改版本中进行修复。该 SQL 语句执行完成后，`renderNotes` 会再次运行，更新显示内容。

在 renderNotes 函数中，我们又有一个事务。但请注意，事务变成了只读事务。可以看到，我们查出了所有行，并按照列 updated 排序。这样，我们就可以总是最先取到最新的数据。这里暂且忽略空数组参数。SQL 执行完成后，我们就可以使用查询结果了。事务对象本身和查询结果作为参数传递给了 executeSql 的 success 事件处理器。results 对象是 SQLResultSet 的一个实例。它有一个 rows 属性，而该属性又有一个 length 属性，我们可以使用它作为循环条件遍历结果集。为了获取一行数据，我们使用相应的行号作为参数调用了 item 方法。那个行对象是一个键 / 值对集合，代表了行中的列。我们使用一个字符串变量构造表格（是的，是的，我知道，表格已经过时了），然后追加到 DOM 上。图 5-2 展示了界面的样子。（是的，界面设计可以更美观。）

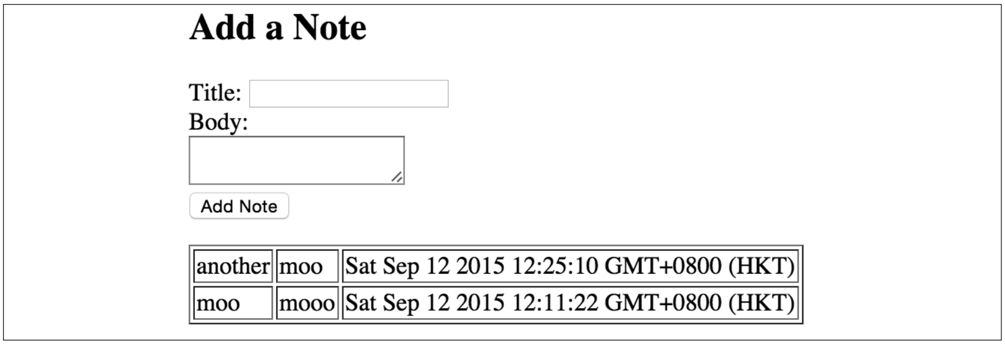


图 5-2：笔记表单

好了，既然你已经对 Web SQL 的工作原理有了基本的认识，那就让我们重点看一下示例 5-3 中的插入语句。

```
tx.executeSql("insert into notes(title,body,updated) values(" +
"'" + title + "','" + body + "'," + (new Date().getTime()) +")");
```

在执行时，该语句会生成类似下面这样的 SQL 语句。

```
insert into notes(title, body, updated)
values('some title', 'some body', 1)
```

以上代码的问题是，如果表单值本身包含一个单引号字符，则 SQL 执行就会中止。通常，允许用户输入参与动态 SQL 创建会导致所谓的 SQL 注入攻击。这很讨厌，所幸很容易修复。还记得第二个空数组参数吗？你可以使用 SQL 中的“标记”表示变量，而不是通过连接创建一个动态 SQL 字符串，然后使用数组参数提供那些值。这很容易实现，示例 5-4 对此进行了演示。

示例 5-4　test4.html 的一部分

```
function addNote(e) {
    e.preventDefault();
    //没有验证
```

```
        var title = $title.val();
        var body = $body.val();

        db.transaction(function(tx) {
            tx.executeSql("insert into notes(title,body,updated) "+
            "values(?,?,?)", [title, body, new Date().getTime()]);
        },dbError,function(tx) {
            $title.val("");
            $body.val("");
            renderNotes();
        });
    }
```

注意，现在 SQL 成了一个多么简单的字符串——没有嵌入任何变量。之前变量所在的位置，现在都成了问号。它们会被下一个参数（即数组）所包含的值按同样的顺序替换。

5.6 使用开发者工具查看Web SQL

Chrome 开发者工具提供了很好的 Web SQL 支持。可以看到，在资源选项卡中，Web SQL 有专门的部分，其中列出了所有已定义的数据库。选中一个并展开后，就可以选择一张表来查看其中所有的数据了（如图 5-3 所示）。

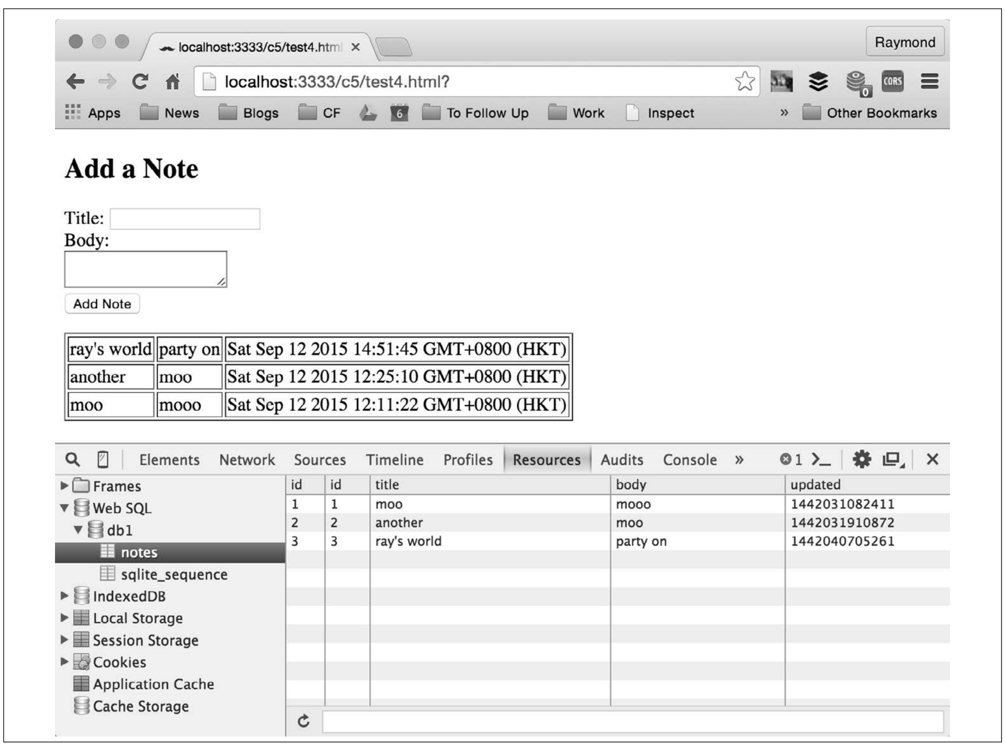


图 5-3：Chrome 的 Web SQL 视图

数据下方有一个空文本框，你可以在此输入列名对视图进行过滤，只保留所输入的列和主键。有一项功能比较隐蔽，就是点击数据库本身会出现一个控制台，你可以在那里任意输入 SQL 语句（如图 5-4 所示）。

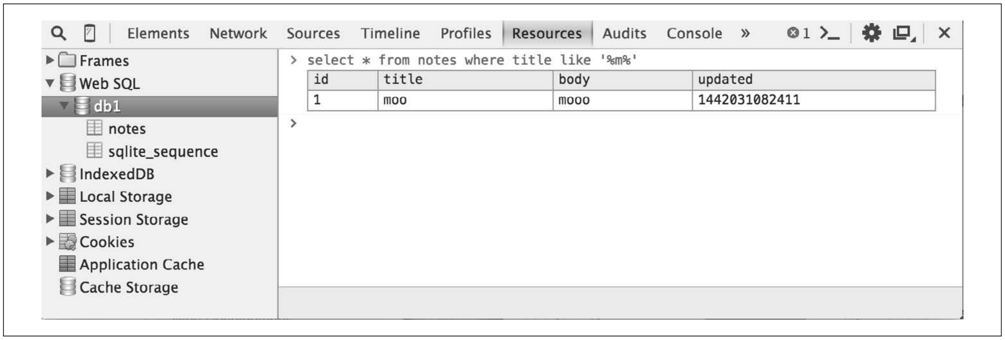


图 5-4：在开发者工具中运行 SQL 命令

正如本章开头所说，你或许不会在新项目中使用 Web SQL，但如果不得不调试已有的程序，Chrome 开发者工具就会非常有用。

5.7 浏览器支持和使用建议

让我们看一下当前 Web SQL 的浏览器支持情况，如图 5-5 所示。

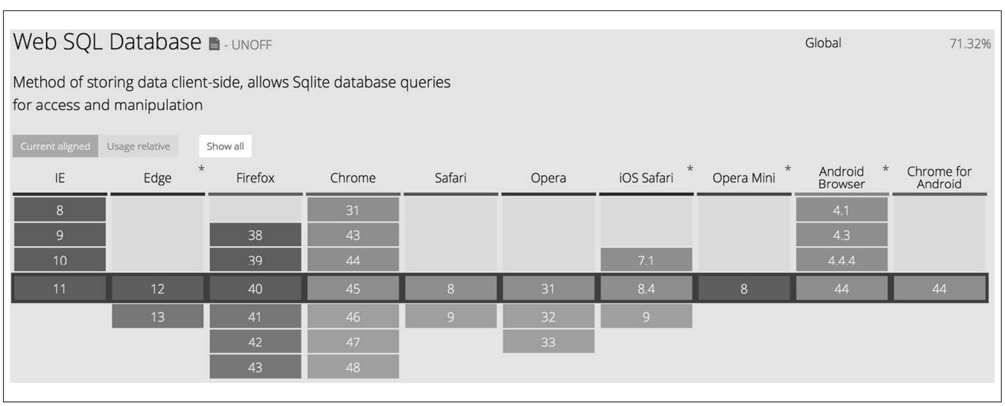


图 5-5：来自 CanIUse.com 的 Web SQL 浏览器支持报告

可以看出，Chrome、Safari 及相应的移动版本支持这项特性。对于一个已经废弃的规范而言，这已经不错了。但是可惜，它真的废弃了（或者说即将废弃）。因此我建议，如果可以避免，那就不要使用它。如果你想使用它，那么可以使用 IndexedDB 的地方当然也适用于这里。

第 6 章

使用库简化客户端存储

6.1 “使用库，卢克……”

好吧，这句话并不是原话[1]，但请考虑这一建议。客户端存储是现代浏览器提供的一个实用的特性。有鉴于此，乐于助人的开发人员创建了可以简化客户端存储的库。有时候，这些库让 API 更易用；有时候，它们增加了连原生 API 都不支持的功能。和你想的一样，有许多这样的库可以供你使用，但本章将着重介绍 Lockr、Dexie 和 localForage 这三个库。

6.2 使用Lockr

我们要介绍的第一个库是 Lockr，它封装了 Web 存储 API（如图 6-1 所示）。你可能马上就想知道究竟为什么要简化 Web 存储。跟随我的思路，你一会儿就知道原因了。Lockr 提供了一个类似 Redis 的 Web 存储 API。不过，你不必因为从来没有听说过 Redis 而担心。这是一个很小的库（2.5KB），和本章将要讨论的其他库一样，它是开源且免费的。Lockr 的主页为：https://github.com/tsironis/Lockr。

注 1：此处，作者套用了科幻系列电影《星球大战》中的一句话：“使用原力，卢克。”在电影中，卢克是绝地武士和原力使用者。——编者注

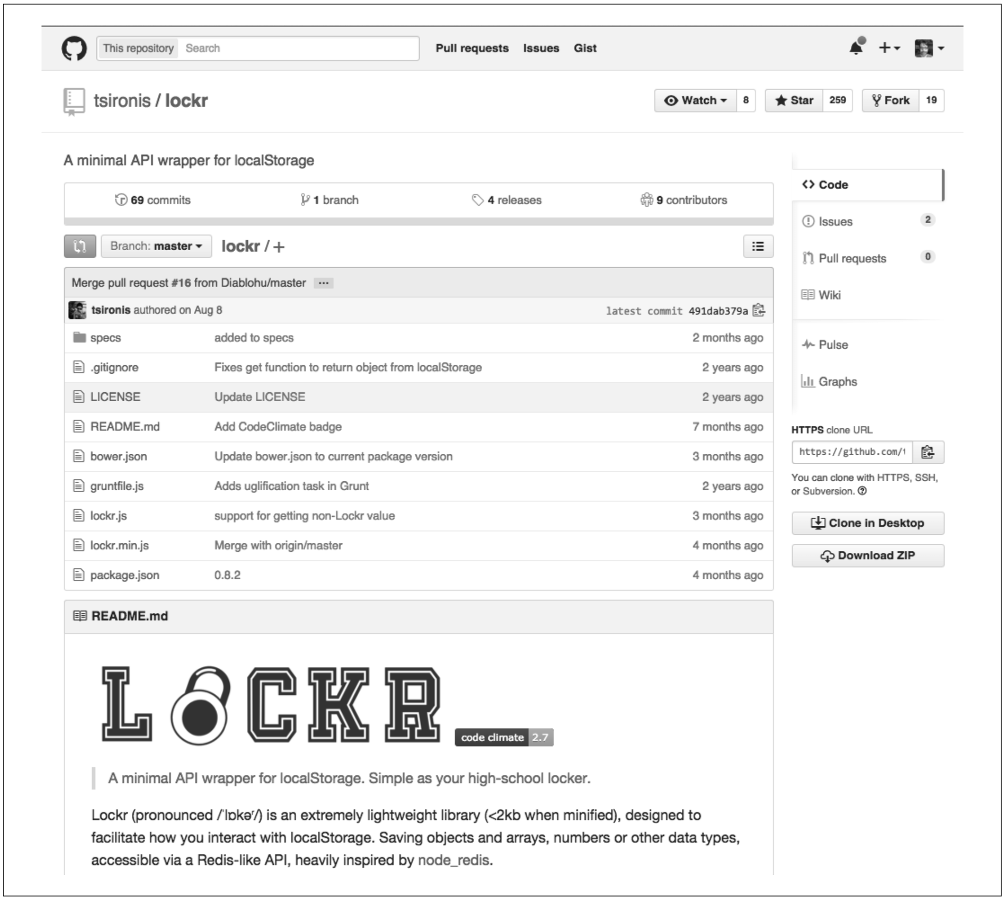


图 6-1：Lockr 的 GitHub 主页

你可以从 GitHub 获取源代码或者通过 Bower 安装：`bower install lockr`。下载完成后，只需像包含其他任何 JavaScript 库一样，把它包含到代码中。

Lockr 的使用相对简单。例如，下面的语句设置了一个值：

```
Lockr.set("name", "Raymond");
```

而下面的语句获取了一个值：

```
Lockr.get("name");
```

那么为什么还要费事讨论它呢？我们来看两个有趣的例子。首先，看一下示例 6-1。

示例 6-1　lockr/test1.html

```
<!doctype html>
<html>
```

```
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="lockr.min.js"></script>
</head>

<body>

<script>

$(document).ready(function() {

    Lockr.set("name", "Ray");
    Lockr.set("age", 43);

    var name = Lockr.get("name");
    var age = Lockr.get("age");
    console.log(name, age + 1);

    //与localStorage对比
    localStorage.setItem("age_ls", 43);
    console.log(localStorage.getItem("age_ls")+1);

});

</script>
</body>
</html>
```

以上代码首先做了两个简单的设置——名字和年龄，然后获取它们的值并在控制台显示。不过请注意，我们将 `age` 的值加了 `1`。紧接着是一个类似的测试，使用了“常规”的 Web 存储。运行这段代码，你会发现有趣之处，如图 6-2 所示。



图 6-2：比较 Lockr 和基本的 Web 存储

看到了吗？使用 Lockr 获取并修改数值的时候，可以获得正确的结果。但 Web 存储把什么都当作字符串处理。因此，获取年龄并“加 1”的结果是将 `1` 追加到了值的末尾。好，这没什么大不了，但示例 6-2 又如何？

示例 6-2 lockr/test2.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="lockr.min.js"></script>
```

```
</head>

<body>

<script>

$(document).ready(function() {

    Lockr.set("stuff", [1,2,3,4]);
    Lockr.set("person", {
        name:"Ray",
        age:43,
        hobbies:["stuff","more stuff"]
    });

    var stuff = Lockr.get("stuff");
    var person = Lockr.get("person");
    console.dir(stuff);
    console.dir(person);

});

</script>
</body>
</html>
```

在这个示例中，我们使用 Lockr 获取并存储了两个复杂对象。在存储或检索时，我们都没有把它们序列化为 JSON，如图 6-3 所示。



图 6-3：Lockr 轻松处理复杂数据

但等一下，它还可以更好。当 Web 存储中没有值的时候，你还可以让 Lockr 的 `get` API 返回一个默认值，如下所示。

```
var coolness = Lockr.get("coolness", "Infinity!");
```

在这段代码中，如果 Web 存储中没有存储 `coolness` 的值，则会返回 `"Infinity!"`。（关于这一点，lockr/test3.html 提供了一个完整的演示程序。）

Lockr 还支持 `hash` 这种特殊的值类型。Lockr 允许你向数组中添加唯一值。如果你试图添加的值已经存在，那它就不会再次被添加。例如，假定有一个包含三个值的数组 `[1, 8, 9]`，如果你试图再添加一个 `9`，那么 Lockr 不会把它追加到数组中；而如果你试图添加 `4`，那么 Lockr 就会允许，数组则会变成 `[1, 8, 9, 4]`。Lockr 通过 `sadd` API 实现这一操作，如示例 6-3 所示。

示例 6-3　lockr/test4.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="lockr.min.js"></script>
</head>

<body>

<script>

$(document).ready(function() {
    Lockr.set("testS", []);

    Lockr.sadd("testS",1);
    Lockr.sadd("testS",2);
    Lockr.sadd("testS",3);
    Lockr.sadd("testS",2);
    Lockr.sadd("testS",2);
    Lockr.sadd("testS",1);

    console.log(Lockr.get("testS"));
    console.log(Lockr.smembers("testS"));

    Lockr.srem("testS", 3);

    console.log(Lockr.smembers("testS"));

    console.log(Lockr.sismember("testS", 3));

});

</script>
</body>
</html>
```

首先，用一个空数组对键 `testS` 进行初始化。然后，使用 `sadd` 方法添加一些值。不过，在所有那些值都添加完以后，数组中只有 1、2、3 三个数据项。你还可以使用 `smembers` 方法返回所有的值。

接下来，使用 `srem` 方法删除一个值。当 `smembers` 方法再次返回时，就只剩下 `1` 和 `2`。最后，`sismember` 会根据一个值在 `hash` 中是否存在，返回 `true` 或 `false`。

总之，Lockr 是一个相当优秀且小巧的库。虽然 Web 存储已经很容易使用了，但单单是 Lockr 在数据处理方面的功能就足以让我感兴趣了。如果再考虑到它令人难以置信的小巧程度，这个库就更有吸引力了。

6.3 使用Dexie简化IndexedDB

接下来，我们将看一下 Dexie（如图 6-4 所示）。相对于略显复杂的 IndexedDB API 而言，该封装器要简单许多。和本章讨论的所有库一样，它是 100% 免费且开源的。http://www.dexie.org 提供了有关该库的更多信息及下载。

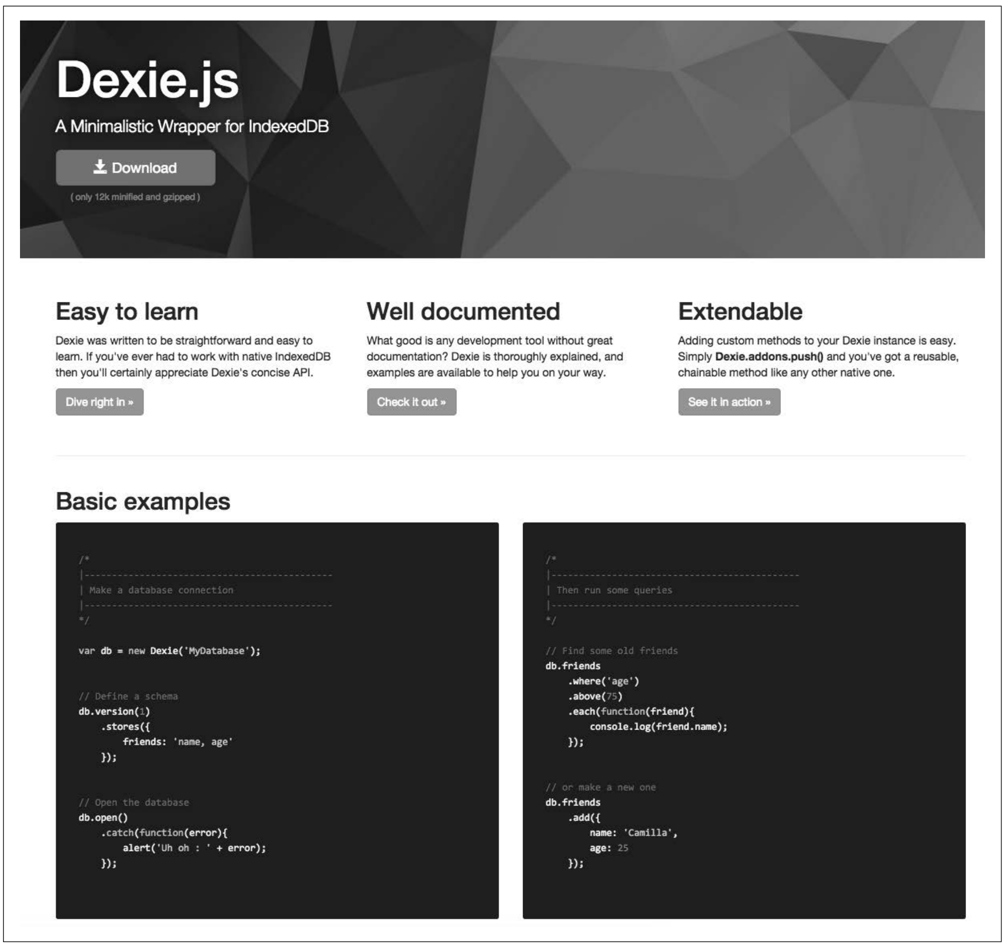


图 6-4：Dexie 网站

对于 Dexie，有三种安装方法。你可以使用 Bower（`bower install dexie`）、npm（`npm install dexie`），或者直接从 GitHub 上下载。

在将该库加载到 Web 页面之后，你会发现 Dexie 的使用非常简单。例如，下面的代码创建了一个 IndexedDB 数据库指针，并用一个名为 `notes` 的对象存储对它进行了初始化。

```
var db = new Dexie("name-here");
db.version(1).stores({
    notes:'text,created'
});
db.open();
```

你可能已经猜到，字符串 `'text,created'` 是期望存储的数据属性。不过，Dexie 更进一步，它允许你通过简单的标记定义这些属性的行为。例如，下面这个版本增加了一个自增键 `id`。

```
var db = new Dexie("name-here");
db.version(1).stores({
    notes:'++id,text,created'
});
db.open();
```

示例 6-4 完整地演示了该特性在实际中的应用。

示例 6-4　dexie/test1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="Dexie.min.js"></script>
</head>

<body>

<script>

$(document).ready(function() {


    var db = new Dexie("dexie1");
    db.version(1).stores({
        notes:"++id,text,created"
    });
    db.open();

    console.dir(db);

});

</script>
</body>
</html>
```

总的来说，该示例并没做多少工作，它只是创建了一个经 Dexie 封装的数据库实例，`console.dir` 也没有多大用处，但如果使用浏览器开发者工具查看 IndexedDB 实例，你就会看到一个新创建的数据库实例 `dexie1`（如图 6-5 所示）。

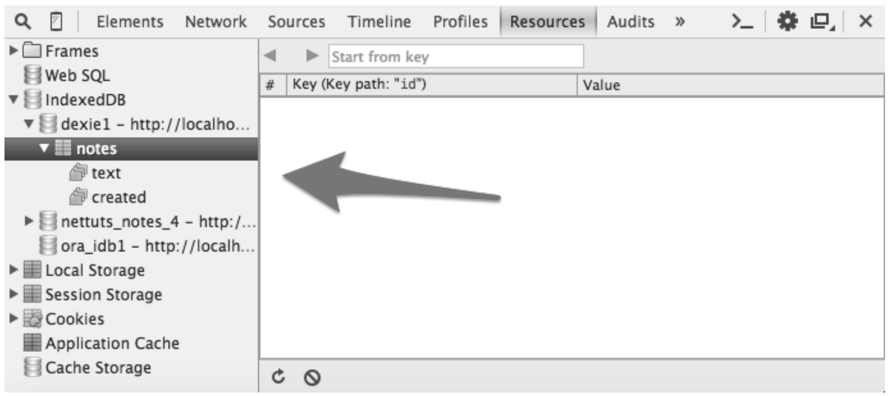


图 6-5：毫不费力地新建一个 IndexedDB！

目前为止，一切顺利，但让我们看几个基本的 CRUD 操作示例。下面的代码展示了如何添加数据。

```
db.notes.add(
{ text:'foo', created:new Date().getTime() }
).then(function() {
 console.log('Note added.');
}).catch(function(err) {
});
```

如果你熟悉 Promise，那么就会觉得这种语法看起来很眼熟。毫无疑问，它比你平常使用的事务 API 更简单。（要知道，Dexie 在后台仍然会使用事务，只是你在进行简单操作时不必考虑了。如果需要使用 Dexie 完成多个 CRUD 操作，那么也还是有一个事务 API 可以使用。但是，在这个简短的介绍中，我们不会涉及这个话题。没错，它仍然要比 IndexedDB 预置的 API 更简单。）

那么数据读取呢？没错，同样很简单：

```
db.notes.get(1).then(function(note) {
  console.dir(note);
});
```

更新稍微复杂一些——你要将正在更新的对象的键作为参数传入：

```
db.notes.put(
{ text:'foo', created:new Date().getTime(), key }
).then(function() {
```

```
  console.log('Note updated.');
}).catch(function(err) {
});
```

更棒的是，你可以使用 put 方法而不传入主键。此时，它会执行插入而不是更新。这让你可以只使用 put 方法就实现插入和更新两种操作，而不必在它和 add 之间切换。

最后是删除操作：

```
db.notes.delete(1).then(function(note) {
  console.log("Removed");
});
```

让我们修改之前的例子，向数据库添加一点儿数据（示例 6-5）。

示例 6-5　dexie/test2.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="Dexie.min.js"></script>
</head>

<body>

<script>

$(document).ready(function() {

    var db = new Dexie("dexie1");
    db.version(1).stores({
        notes:"++id,text,created"
    });
    db.open();

    db.notes.add(
        { text:"foo",created:new Date().getTime() }
    ).then(function() {
         console.log("Note added.");
    }).catch(function(err) {
        console.dir(err);
    });

});

</script>
</body>
</html>
```

现在，我们的模板实际地添加了一点儿数据。通常，你会将这个过程和一个表单联系起来，但如果你在浏览器中打开这个示例，并查看开发者工具，那么就可以看到新增的数

据，如图 6-6 所示。

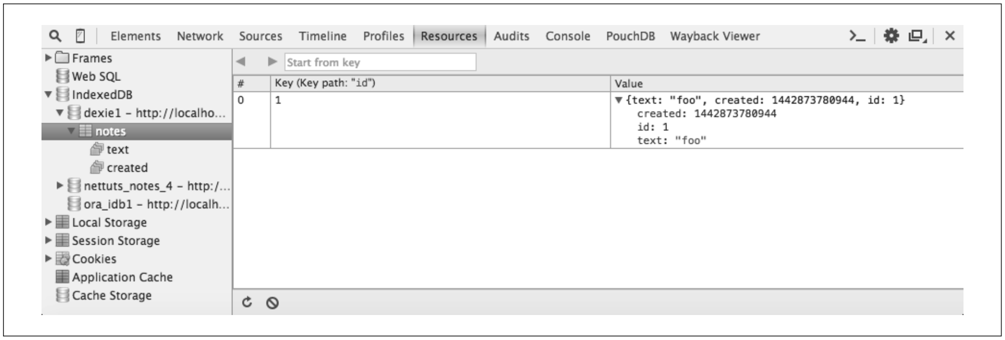


图 6-6：通过 Dexie 新增的数据

最后一个难点是搜索数据，而这才是 Dexie 真正的亮点所在。让我们看一个简单的例子——查询某个特定列（或属性）的值低于目标值的数据。

```
db.something.where("column").below(value).each(
function(item) {
  console.log('runs for each match')
});
```

你可能想查询高于目标值的数据，而不是低于目标值：

```
db.something.where("column").above(value).each(
function(item) {
  console.log('runs for each match')
});
```

或者介于两个值之间：

```
db.something.where("column").between(value1, value2).each(
function(item) {
  console.log('runs for each match')
});
```

示例 6-6 简单地演示了这个 API。

示例 6-6　dexie/test3.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="Dexie.min.js"></script>
</head>

<body>
```

```
<script>

$(document).ready(function() {

    var db = new Dexie("dexie3");
    db.version(1).stores({
        people:"email,name,age"
    });
    db.open();

    db.people.put({ email:"raymondcamden@gmail.com", name:"Raymond", age:43 });
    db.people.put({ email:"elric@google.com", name:"Elric", age:23 });
    db.people.put({ email:"zula@google.com", name:"Zula", age:12 });

    db.people.where("age").between(20,50).each(function(person) {
        console.log("age match",JSON.stringify(person));
    });

    db.people.where("name").anyOf(["Elric","Zula"]).each(function(person) {
        console.log("name match",JSON.stringify(person));
    });
});

</script>
</body>
</html>
```

运行这个示例，你可能会注意到输出有些有趣（如图 6-7 所示）。

```
14:54:02.709    age match {"email":"elric@google.com","name":"Elric","age":23}
14:54:02.710    name match {"email":"elric@google.com","name":"Elric","age":23}
14:54:02.711    age match {"email":"raymondcamden@gmail.com","name":"Raymond","age":43}
14:54:02.711    name match {"email":"zula@google.com","name":"Zula","age":12}
```

图 6-7：Dexie 搜索示例的输出

虽然结果是正确的，但不要忘记，它们是异步生成的。这就是为什么你在控制台中看到的结果是“混合”的。虽然这未必和本书有关，但我们还是快速介绍一下如何处理类似的情况。前面已经提到过，Dexie 支持事务，而且事务本身知道自己何时结束。示例 6-7 是前例的一个修改版本，它使用事务等待查询操作完成。

示例 6-7　dexie/test4.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="Dexie.min.js"></script>
</head>
```

```
<body>

<script>

$(document).ready(function() {


    var db = new Dexie("dexie3");
    db.version(1).stores({
        people:"email,name,age"
    });
    db.open();

    db.people.put({ email:"raymondcamden@gmail.com", name:"Raymond", age:43 });
    db.people.put({ email:"elric@google.com", name:"Elric", age:23 });
    db.people.put({ email:"zula@google.com", name:"Zula", age:12 });

    var ageResults, anyResults;
    db.transaction('r', db.people, function() {
        ageQuery = db.people.where("age").between(20,50).toArray().then(
        function(age) {
            ageResults = age;
        });
        anyQuery = db.people.where("name").anyOf(["Elric","Zula"]).toArray().then(
        function(any) {
            anyResults = any;
        });

    }).then(function() {
        console.log(JSON.stringify(ageResults));
        console.log(JSON.stringify(anyResults));
    });

});

</script>
</body>
</html>
```

还请注意，我们修改了第二个查询，以使用辅助函数 `toArray`。这是 Dexie 提供的一个工具，可以将查询结果返回到一个简单的数组中，这意味着你不需要使用 `each` 方法对结果进行遍历。

6.4 使用localForage

我们要介绍的最后一个库（但别忘了，还有更多的库！）是 localForage（如图 6-8 所示），它是一个由 Mozilla 开源的项目。Firefox localForage 背后是各种客户端存储封装器，支持 IndexedDB、Web SQL 和本地存储。它可以动态地选择最佳的存储机制，将数据存储到用户的浏览器中。localForage 的 GitHub 主页是：https://github.com/localForage/localForage。

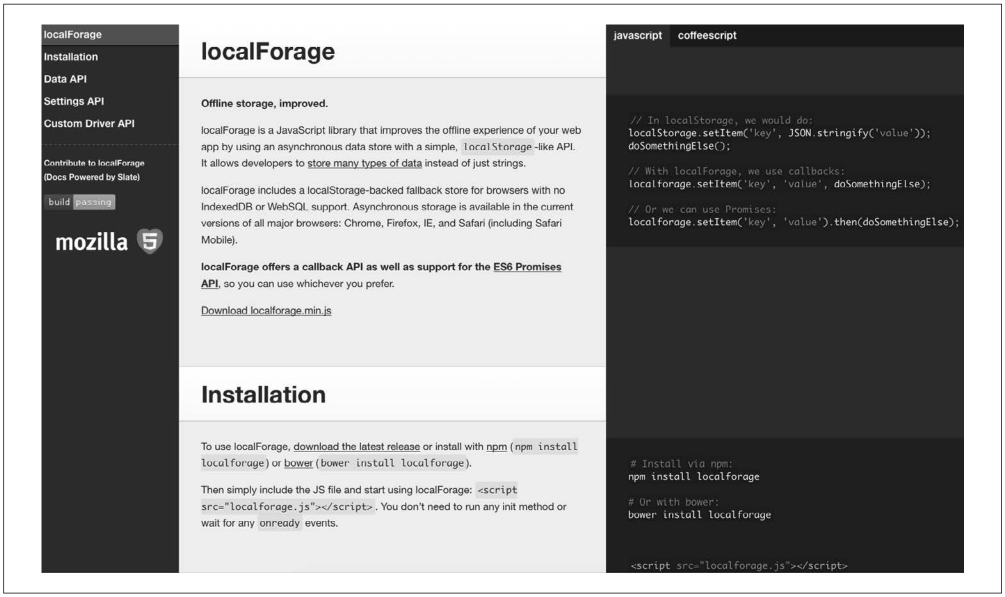


图 6-8：localForage 网站[2]

和本章介绍的其他库一样，有多种方法可以安装 localForage。你可以使用 Bower（`bower install localforage`）、npm（`npm install localforage`），或者直接从 GitHub 上下载代码。

localForage 的 API 是完全异步的，但既支持“老式”的回调，也支持“新兴”的、基于 Promise 的 API。你可以使用自己最习惯的方式。

下面这个简单的例子展示了如何设置一个值，并在这个值持久化之后执行回调函数。

```
localforage.setItem("name", value, function(err, value) {
});
```

下面是 Promise 风格的版本。

```
localforage.setItem("name", value).then(function(value) { });
```

以上两段代码所做的操作完全相同（是的，从技术上讲，第二个示例需要调用 `catch`），因此，你可以使用任何在你看来更简单的方式。检索一个值的情况相同，如下所示。

```
localforage.getItem("name", function(err, value) { });
localforage.getItem("name").then(function(value) { });
```

示例 6-8 简单地展示了实际的读写操作。

注 2：该截图源自 https://mozilla.github.io/localForage，但似乎此网页已不存在。关于 localForage 的更多信息，请见它的 GitHub 主页：https://github.com/localForage/localForage。——编者注

示例 6-8　localForage/test1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="localforage.js"></script>
</head>

<body>

<script>

$(document).ready(function() {


    localforage.setItem("name", "Ray", function(err, value) {
        if(err) console.dir(err);
        console.log(value);
    });

    localforage.setItem("age", 43, function(err, value) {
        if(err) console.dir(err);
        console.log(value);

        localforage.getItem("age").then(function(value) {
            console.log("the value of age plus one is "+(value+1));
        });
    });

});

</script>
</body>
</html>
```

第一个示例只是将 `name` 的值设为 `Ray`。数据存储成功后会激活回调函数。结果回显到控制台上并没什么用，因为它（显然）和我们传入的内容相同。第二个示例存储了一个数值，为了确保它被正确地存储，我们获取这个值并将该值加 `1`。

localForage 提供的 API 还包括删除数据（`removeItem`）、清理存储（`clear`）、计算键的数量（`length`）以及获取所有的键（`keys`）。你还可以通过一个简单的 `iterate` 调用，遍历所有的键 / 值对。

```
localforage.iterate(function(value, key, index) {

}, callback);
```

如前所述，localForage 会设法使用当前浏览器上“最好”的存储方法。在默认情况下，localForage 会首先尝试 IndexedDB，然后是 Web SQL，最后是本地存储。最酷的是，你居

然可以规定一个不同的优先级。`setDriver` API 让你可以指定希望使用的系统，或者按顺序指定一组系统。下面是一个例子，它优先选择 Web SQL，其次是 IndexedDB。

```
localforage.setDriver(localforage.WEBSQL, localforage.INDEXEDDB);
```

注意，本地存储不需要指定。如果 localForage 无法找到优先选择的存储系统，则会自动采用默认设置。

localForage 没有提供任何查询或搜索功能，在采用这个特定的库之前要牢记这一点。这意味着 localForage 更适合简单的存储需求，比如你希望通过键而不是某种特殊查询来检索的大型数据集。

6.5 更多选择

我们已经说过多次，本章只介绍了少数几个客户端存储库。下面是几个其他的库，也许你会希望了解一下。

- PouchDB（http://pouchdb.com）是一个功能非常强大的可选方案。实际上，它是如此强大，以至于我都担心在本章的末尾对它进行如此简短的介绍太过分了。该库的开发者已经在客户端存储领域做了大量的工作，他们都是这个领域的知名专家。PouchDB 可能吸引你的最大的特点是支持数据同步。
- lawnchair（http://brian.io/lawnchair）是一个比较“古老”的库，也是通过适配器 API 支持多种存储方法。
- 最后，你还可以仔细研究一下由 Juho Vepsäläinen 创建的这个实用的列表（https://github.com/bebraw/jswiki/wiki/Storage-libraries）中的库。

第 7 章

构建示例应用程序

7.1 让我们构建真实的应用程序！

你已经了解了多种客户端存储技术，以及一些让它们更容易使用的库。现在，让我们使用其中的部分技术构建一个简单但真实的应用程序。我们将要构建的是一个在某公司（即将在纽约证券交易所上市的“卡姆登公司”[1]）内网使用、供用户查找同事的工具。该应用程序可以使用传统的应用程序服务器模型构建，但我们决定使用现代 Web 标准使其别出心裁。为了实现准实时搜索，我们将使用客户端存储，在用户的浏览器中保存员工数据库的一个副本。当然，这将引出各种有趣的问题。

首先，我们该如何处理同步？公司不是一成不变的，总是会有人加入或离开。当然，人员变动频率取决于公司本身。但显然，必须考虑某种形式的策略，使用户的数据副本与服务器上的真实列表保持同步。所幸，在这个场景中，我们不必考虑用户编辑。服务器端总是“真实”的，也就是说，同步的时候可以不考虑客户端的变化。对于这个演示程序，我们根本不必担心同步。但在真实世界中，应用程序服务器会提供一个 API，客户端可以通过它告诉（当然，我这里说的告诉是指通过代码）服务器：“我的数据副本最后一次更新是在 2015 年 10 月 10 日上午 8:55。”服务器就可以使用这个日期之后发生的一系列更改进行响应。那些更改会涵盖删除操作（有人离开了公司）、修改操作（有人改名或者获得了新头衔）和添加操作（聘用了新员工）。客户端代码会应用那些更改，然后记录当前时间。这样，下次向服务器请求数据时，就可以正确地接收到更改后的数据。

注 1：这是作者用自己的姓氏开的玩笑。——编者注

下一个问题有点棘手：隐私。对于你不希望分享的信息，比如工资，公司的数据库可能做了很好的处理。请记住，我们实际上是将私人信息发送给了每名员工。虽然你可能信任他们，但仍然不能将员工的隐私暴露在风险之中。一个可能安全的标准是“只分享名片上的信息”，但在这一点上，无疑需要十分谨慎。要知道，你无法在客户端“过滤”不安全的数据。如果你的应用服务器正在返回私人数据，那么任何人打开浏览器的开发者工具都可以清楚地看到它。用户可以打开并查看浏览器获得的任何数据。通常，我个人习惯在浏览网页时开着浏览器工具，并会出于好奇而本能地查看 AJAX 调用和数据。我是“好人”，但你必须假设“不像我那么好的人”也会查看这些内容。

最后一个问题是性能。假如有一个一万人的“小”公司，你如何高效地将那些数据传输到浏览器？我们已经说过，这里假定的场景是公司内网，这就相当于已经假定是在桌面或局域网环境里，但你会希望知道将要发送到客户端的数据包的大小。本章稍后会讨论一种应对这种情况的方法。

好了，让我们聊聊数据！

7.2 示例数据

为了让事情尽可能简单，我们的“服务器”只是一个简单的 JSON 数据文件。如上所述，我们不会使用同步及创建更新。因此，使用一个 JSON 平面文件就可以很好地满足我们的需求了。为了让事情更简单，我们将使用一个很酷的免费 API 生成数据：随机用户生成器（https://randomuser.me），如图 7-1 所示。

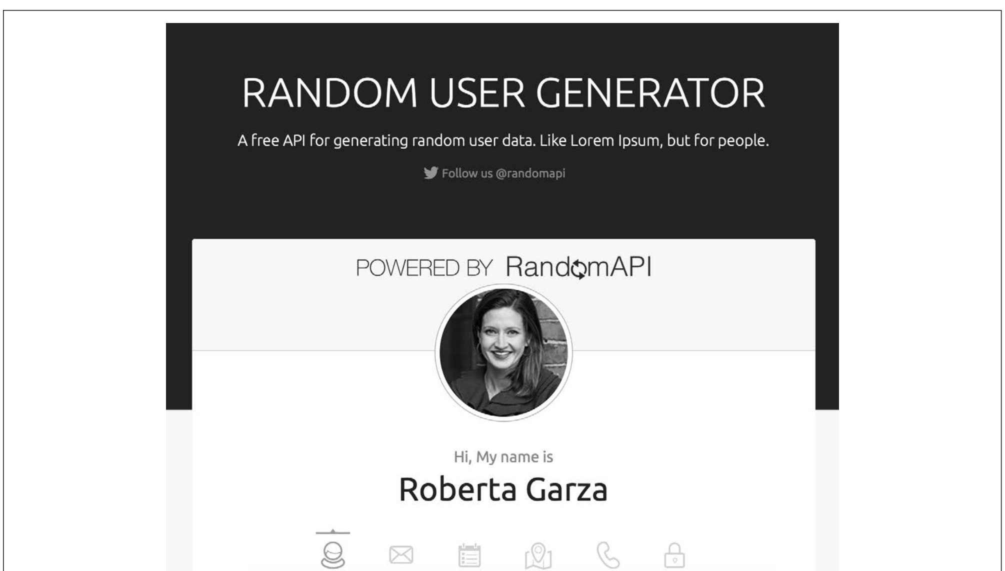


图 7-1：随机用户生成器

该网站提供了一个返回用户信息的免费 API。这些用户信息包含许多细节，可以用在我们这里要构建的演示程序中。示例 7-1 是从他们的文档中摘录的一个输出样例。

示例 7-1　API 返回结果样例


```
{
  results: [{
    user: {
      gender: "female",
      name: {
        title: "ms",
        first: "manuela",
        last: "velasco"
      },
      location: {
        street: "1969 calle de alberto aguilera",
        city: "la coruña",
        state: "asturias",
        zip: "56298"
      },
      email: "manuela.velasco50@example.com",
      username: "heavybutterfly920",
      password: "enterprise",
      salt: ">egEn6YsO",
      md5: "2dd1894ea9d19bf5479992da95713a3a",
      sha1: "ba230bc400723f470b68e9609ab7d0e6cf123b59",
      sha256: "f4f52bf8c5ad7fc759d1d415e508aa0b7946d4ba",
      registered: "1303647245",
      dob: "415458547",
      phone: "994-131-106",
      cell: "626-695-164",
      DNI: "52434048-I",
      picture: {
        large: "http://api.randomuser.me/portraits/women/39.jpg",
        medium: "http://api.randomuser.me/portraits/med/women/39.jpg",
        thumbnail: "http://api.randomuser.me/portraits/thumb/women/39.jpg",
      },
      version: "0.6"
      nationality: "ES"
    },
    seed: "graywolf"
  }]
}
```


虽然该 API 相当地简单易用，但我们希望在演示程序中使用一个静态数据集。如果你注册了 RandomAPI（http://www.randomapi.com），就可以使用随机用户 API 获取多达 10 000 条结果。RandomAPI 网站——不难想象——是一个提供随机数据的 API 集合。总之，这两个网站确实都极其有用，你也可以在自己的应用程序中使用它们。在构建应用程序时，使用“切合实际”的随机数据是一种很棒的方法。

针对这个演示程序，我注册并申请了 10 000 条用户数据，并将数据保存在名为 users.json 的文件中。你可以在包含本书示例代码的 zip 文件中找到它，路径为 c7/data。本章前面已

经讨论过，对于应用程序暴露什么数据，要格外小心。因为我们仅仅是按原样获取了随机用户数据，所以其中肯定会有一些我们绝对不想分享的信息。不仅如此，对于数据中的用户信息，我们的演示程序大约只会使用一半。这意味着，从服务器发送到前端的数据有许多就浪费了。在准备交付的应用程序中，我们对所有这些都要非常了解。但是，回想一下我们已经完成的工作。我们已经将 API 进行了简化，只保留了可以满足应用程序需求的基本信息。还可以做点其他的什么事情，来加快数据加载吗？

一个简单的方法是使用 GZip 压缩。Web 服务器可以使用这个设置，在数据资产发送给浏览器之前，对其进行 zip 压缩。Web 服务器足够智能，只有当浏览器在头信息中告诉它自己支持这个特性时，它才会使用这个特性。而且，由于几乎所有的现代浏览器都支持这个特性，因此很容易用它加速数据传输，尤其是 Apache 让这个功能的启用变得非常简单。它能够提供多大的帮助呢？

users.json 文件的大小为 13.5MB，这不算小。即使是没有经过充分优化的图片，也多半不会超过 1MB。因此，下载那个文件会严重地影响性能。图 7-2 显示了在使用任何压缩技术之前，通过浏览器请求文件时，Chrome 报告的 JSON 文件大小。

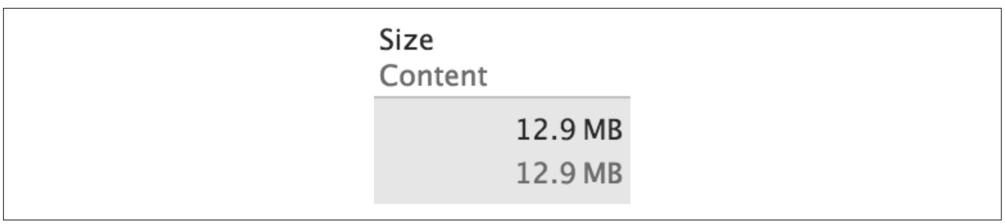


图 7-2：Chrome 的文件请求报告

图 7-3 显示了启用 Apache 压缩功能以后的文件大小。

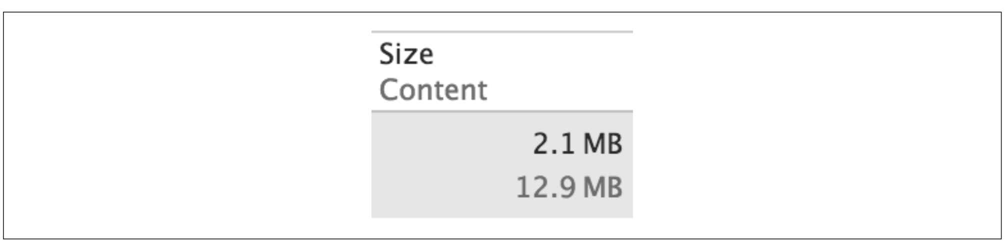


图 7-3：没错，更小了

差别相当惊人。记住，浏览器还要在客户端解压那个文件，所以使用这个方法时要谨慎。我仍然不建议通过网络发送超过 10MB 的数据。至少在我们的例子里，这是起初“最坏情况下”的负载，而后续调用（再次使用一个假想的应用程序服务器）将只发送有变化的数据。

现在，我们已经看过了数据，接下来让我们看一下已完成的应用程序。

7.3 应用程序

当你第一次点击该应用程序时，它会获取初始数据集（那个很大的 JSON 文件），并开始将其插入本地数据存储。由于这需要一点时间，因此我们使用一个模态窗口告诉用户发生了什么。对于这个应用程序，该窗口相当简单——只有一条消息（如图 7-4 所示）。你可以改进这条消息，用它报告应用程序是否正在下载初始数据，或者是否已经开始将其插入本地存储。

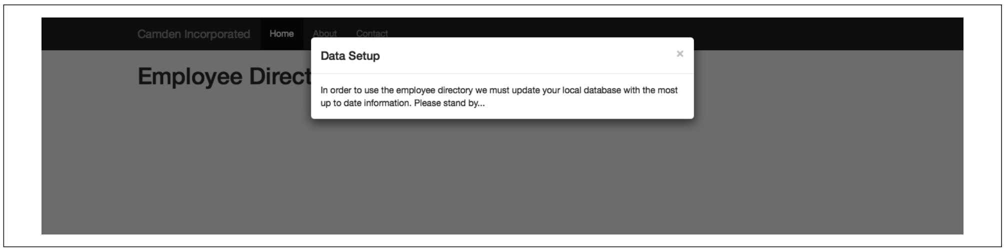


图 7-4：应用程序设置过程中显示的消息

当所有数据都加载完成后，用户会看到一个基本的表单，如图 7-5 所示。在这个应用程序中，你只能使用名和姓进行搜索。

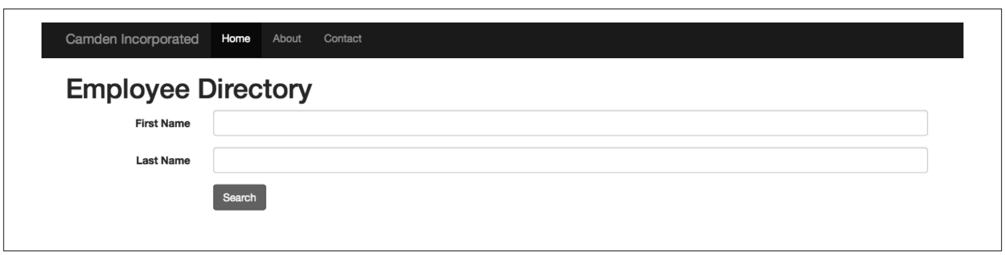


图 7-5：搜索框

接下来，你可以开始搜索了。你可以只搜索名或姓，也可以两者同时搜索。图 7-6 展示了部分搜索结果。

即使什么也没找到，应用程序也会让你知道究竟发生了什么。总之，这是一个相当简单的界面。后续你可以加入更多的过滤器（比如业务部门或经理），进一步增强搜索功能。也许你会好奇那些图片的出处，它们全都来自随机用户生成器。

我们已经讨论了数据，并展示了应用程序界面。接下来让我们看一下界面背后的代码。

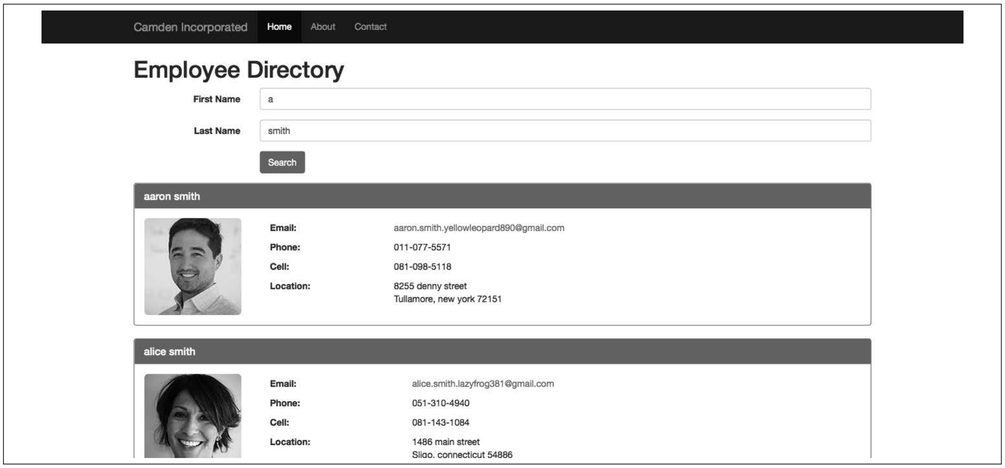


图 7-6：搜索结果

7.4　代码

该应用程序将使用本地存储和 IndexedDB。本地存储只用于记录数据是否已经加载。IndexedDB 将存储数据本身。对于本地存储，虽然这里的用法相当简单，但我们还是会使用 Lockr。对于 IndexedDB，我们将使用 Dexie 简化数据插入及搜索。

首先，让我们看一下应用程序如何确定数据是否已经缓存到客户端。示例 7-2 展示了用来确定本地数据是否存在的函数。

示例 7-2　haveData 函数

```
function haveData() {
    var def = $.Deferred();

    var lastFetch = Lockr.get("lastDataSync");

    if(lastFetch) def.resolve(true);
    else def.resolve(false);

    return def.promise();
}
```

这里有几处比较有趣。函数代码的第一行创建了一个 Deferred 对象，以便可以返回一个 Promise 对象。这让我们可以异步地使用这个函数。不过，我们实际上并没有使用异步过程。我们已经知道，本地存储访问是同步的。但将来，我们可能会更新这个过程，让它变成异步的。调用这个函数的代码并不需要修改。

目前，我们的代码只是使用 Lockr 检查了 `lastDataSync` 属性。如果该属性存在，则表明我们已经有了数据。稍后，我们会将它设置为一个日期值。按照设想，如果你将来把这段代

码连接到一个“真正”的应用服务器上，那么在确定你所需要的新数据时，这个日期会是一条非常有价值的信息。让我们看一下如何调用这段代码（示例 7-3）。

示例 7-3　调用 haveData

```
haveData().then(function(hasData) {

    if(!hasData) {
        console.log("I need to setup the db.");
        setupData().then(appReady);
    } else {
        appReady();
    }

});
```

以上代码调用了 haveData()，并使用返回对象 Promise 的 then 方法进行应用程序设置，它会在数据的异步加载过程完成后执行。没错，数据加载其实不是异步的，但我们已经说过，调用者无需担心那个问题。如果没有数据，这段代码就会激活一个准备数据的调用；否则，它会运行一个函数，表明搜索应用程序已经准备就绪。让我们看一下数据准备函数。

首先，Dexie 需要我们创建一个 IndexedDB 数据库，并定义存储数据的对象存储（示例 7-4）。这是在代码前半部分的 $(document).ready 块内完成的。

示例 7-4　IndexedDB 准备

```
myDb = new Dexie("employee_database");
myDb.version(1).stores({
    employees:"++id,&email,name.first,name.last"
});
myDb.open();
```

前面的章节已经介绍过，Dexie 极大地简化了 IndexedDB 的使用。可以看到，正在创建数据库和对象存储 employees。employees 存储有一个自增的数值型 id、一个建在 email 上的唯一索引以及建在 name.first 和 name.last 上的索引。这些索引创建的依据是我们计划如何查询员工。现在，我们看一下准备数据的函数（示例 7-5）。

示例 7-5　setupData 函数

```
function setupData() {
    var def = $.Deferred();

    //设置modal选项
    $("#setUpModal").modal({
    keyboard: false
});

//现在显示它
$("#setupModal").modal("show");

//现在,获取远程数据
```

```
$.get("data/users.json", function(data) {
    console.log("Loaded JSON, have "+data.results.length+" records.");
    console.dir(data.results[0].user);

    myDb.transaction("rw", myDb.employees, function() {

        data.results.forEach(function(rawEmp) {

            /*
            我们没有原样复制数据,而是稍微修改了一下。
            特别是,原始数据中有一些email或username重复的,
            所以我以这两个字段为基础创建了新的email
            */
            var emp = {
                cell:rawEmp.user.cell,
                dob:rawEmp.user.dob,
                email:rawEmp.user.email.split("@")[0]+"."
                + rawEmp.user.username + "@gmail.com",
                gender:rawEmp.user.gender,
                location:rawEmp.user.location,
                name:rawEmp.user.name,
                phone:rawEmp.user.phone,
                picture:rawEmp.user.picture
            };

            myDb.employees.add(emp);
        });

    }).then(function() {

        //隐藏modal
        $("#setupModal").modal("hide");

        //记录同步完成时间
        Lockr.set("lastDataSync", new Date());

        def.resolve();

    }).catch(function(err) {
        console.log("error in transaction", err);
    });

}, "json");

return def.promise();
}
```

如你所想，这是一个大函数。haveData 函数使用 jQuery Deferred 来处理数据准备过程的异步性。暂且抛开 UI 元素不谈（Bootstrap 让其变得非常简单），最关键的部分始于通过 AJAX 调用获取 JSON 文件。一旦取得该文件，Dexie 就会打开一个事务。对于 JSON 数据中的每个用户，我们需要新建一个对象来进行存储。在理想情况下，数据应该与我们希望存储的数据完全一样，但由于我们使用了从随机用户生成器中获取的数据，因此需要稍作修改。如果你安装了一台应用服务器为代码提供类似的数据，那么你会希望提供的数据

尽可能与需求保持一致。注意，我们稍微修改了邮件地址，因为在随机用户数据中，邮件地址不唯一。这很可能是原 API 的一个 bug，不过，使用 JavaScript 绕过这个问题更容易。然后，添加对象——就是这样。当事务结束时，UI 会再次更新，而前面声明的 Deferred 对象的 `resolve` 方法会被调用。注意，最后一步是使用当前时间更新本地存储，这还是通过 Lockr 实现的。

现在，让我们转到搜索。在数据加载完成或者已经确定数据存在后，`appReady` 函数会被调用，如示例 7-6 所示。

示例 7-6　appReady 函数

```
function appReady() {
    console.log('appReady fired, lets do this');
    //显示搜索表单
    $("#searchFormDiv").show();
    $("#searchForm").on("submit", doSearch);
}
```

这里没有多少内容可以介绍，主要是显示搜索表单，以及注册执行那个搜索的事件处理器。让我们看一下示例 7-7。

示例 7-7　doSearch 函数

```
function doSearch(e) {
    e.preventDefault();
    var fName = $.trim($firstNameField.val());
    var lName = $.trim($lastNameField.val());

    $results.empty();
    console.log('search for -'+fName+'- -'+lName);

    var fnEmps = [];
    var lnEmps = [];
    myDb.transaction('r', myDb.employees, function() {

        if(fName !== '') {
            myDb.employees.where("name.first").startsWithIgnoreCase(fName)
            .each(function(emp) {
                fnEmps.push(emp);
            });
        }

        if(lName !== '') {
            myDb.employees.where("name.last").startsWithIgnoreCase(lName)
            .each(function(emp) {
                lnEmps.push(emp);
            });
        }

    }).then(function() {
        console.log('done');
        var results = [];
```

```
        //只提供了名
        if(fName !== '' && lName === '') {
            console.log('first');
            fnEmps.forEach(function(emp) { results.push(emp); });
        //只提供了姓
        } else if(lName !== '' && fName === '') {
            lnEmps.forEach(function(emp) { results.push(emp); });
        //两者都提供了
        } else {

            //仅返回在两个列表中都存在的对象
            //简单起见,我们为lnEmps中的email值创建一个索引
            //那样,我们就可以在遍历fnEmps时更快地检查它们
            var lnEmails = [];
            lnEmps.forEach(function(emp) { lnEmails.push(emp.email); });

            results = fnEmps.filter(function(emp) {
                return lnEmails.indexOf(emp.email) >= 0;
            });
        }

        //开始渲染结果
        if(results.length) {
            results.forEach(function(r) {
                console.log(r.name.first+' '+r.name.last);
                var result = resultTemplate(r);
                $results.append(result);
            });
        } else {
            $results.html("Sorry, nothing matched your search.");
        }

    }).catch(function(err) {
        console.log('error', err);
    });

}
```

这又是一个大函数，因此，让我们一步一步地分析。首先，获取搜索表单中的当前字段并去掉空格。一旦取得了这些字段，就可以开始搜索了。遗憾的是，我们无法在一次调用中同时搜索两个值，但我们可以在一个事务中完成。所以，我们再次使用 Dexie 提供的函数打开一个事务，然后分别针对名索引和姓索引进行搜索，搜索结果存储在单独的数组中。

事务结束意味着两次搜索（如果只用了一个搜索字段，则为一次搜索）都已经完成。然后，我们需要合并结果。如果没有同时使用两个字段，则情况很简单：所搜索的字段的结果数组会被复制到最终的结果数组。

如果同时使用了两个字段，则情况要稍微复杂一些。我们希望返回在两个结果数组中都存在的结果。我们通过遍历 `lnEmps` 数组，创建一个只包含邮件地址的更简单的数组。然后，遍历 `fnEmps` 数组，并且只接受那些邮件地址在搜索姓的结果中也存在的值。

终于，结果集准备就绪。我们如何展示呢？为了简化将内容动态写出到模板的过程，我们将使用 Handlebars 作为客户端模板语言。Handlebars 让我们可以使用变量标识定义一个模板。我们可以加载这个模板，自动替换标识，然后渲染出 HTML。index.html 定义了处理搜索结果的模板（如示例 7-8 所示）。

示例 7-8　结果模板

```
<script id="result-template" type="text/x-handlebars-template">
<div class="panel panel-primary">
    <div class="panel-heading">
        <h3 class="panel-title">{{name.first}}{{name.last}}</h3>
    </div>
    <div class="panel-body">
        <div class="row">
            <div class="col-md-2">
                <img src="{{picture.medium}}" class="img-rounded">
            </div>
            <div class="col-md-10">
                <table style="width:100%">
                    <tr>
                        <td><b>Email:</b></td>
                        <td><a href="mailto:{{email}}">{{email}}</a></td>
                    </tr>
                    <tr>
                        <td><b>Phone:</b></td>
                        <td>{{phone}}</td>
                    </tr>
                    <tr>
                        <td><b>Cell:</b></td>
                        <td>{{cell}}</td>
                    </tr>
                    <tr valign="top">
                        <td><b>Location:</b></td>
                        <td>{{location.street}}<br/>
                        {{location.city}}, {{location.state}} {{location.zip}}</td>
                    </tr>
                </table>
            </div>
        </div>
    </div>
</div>
</script>
```

可以看到，在上面的代码清单中，每个标识作为一个值都包含在双重花括号（{{　}}）中。Handlebars 可以处理这些标识，并使用搜索出的实际结果数据替换它们。客户端模板语言的好处是极大地简化了从 JavaScript 动态生成输出的过程。

7.5　总结

你可以在下载的 zip 文件中找到演示程序的完整代码。我强烈建议你自己尝试一下，看看可以作些什么修改。你可以添加更多的搜索筛选条件。如果你真的愿意，也可以安装一个应用服务器，着手开发一个“只发送变化内容”的 API。祝你好运！

作者介绍

Raymond Camden 是 IBM 的一名 Developer Advocate，关注 MobileFirst 平台、Bluemix、混合移动开发、Node.js、HTML5 和 Web 标准。他也是一名作家，并在许多会议和用户组中做过各种主题的演讲。欢迎访问他的博客（http://www.raymondcamden.com），关注他的 Twitter 账号（@raymondcamden），或者通过电子邮件（raymondcamden@gmail.com）与他联系。

封面介绍

本书封面上的动物是无纹地松鼠（南非地松鼠）。无纹地松鼠原生于非洲之角的干旱草原和灌木丛。作为地松鼠，它们在地下穴居。

无纹地松鼠有棕色的皮毛，后背颜色深，前面颜色浅。顾名思义，它们背上没有在其他非洲地松鼠属动物身上常见的白色条纹。无纹地松鼠的体重可以长到 0.45 千克，身长可达 25 厘米。此外，它们的尾巴也可以长到 25 厘米。

无纹地松鼠是杂食性动物，树叶、果实、种子和昆虫都是它们的食物。它们大部分时间都在寻找食物，只有在睡觉时才回到巢穴。它们的主要天敌是猛禽、豹、豺和蛇。

O’Reilly 图书封面上的许多动物都已濒临灭绝。对于这个世界而言，它们都很重要。要想了解更多有关如何为它们提供帮助的信息，请访问 animals.oreilly.com。

封面图片来自 Lydekker 的 *Royal Natural History*。

微信连接

回复“前端”查看相关书单

微博连接

关注@图灵教育 每日分享IT好书

QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

客户端存储技术

现代浏览器的一大实用特性是有能力将数据直接存储在用户的计算机或移动设备上。尽管许多人选择将数据迁移至云端，但若使用得当，客户端存储仍然可以帮助Web开发人员节省大量的时间和金钱。本书结合丰富的实例，详解多种客户端存储技术。你将了解如何及何时使用它们、其优缺点以及在应用程序中使用其中一种或多种技术的步骤。

本书还介绍了几种简化客户端存储的开源库，非常适合熟悉JavaScript的Web开发人员。

- 了解不同浏览器对每种客户端存储技术的支持情况
- 使用Web存储（即本地存储）存储列表和偏好设置等简单信息
- 使用IndexedDB存储几乎任何你希望在用户浏览器中存储的信息
- 了解如何为仍旧使用Web SQL的Web应用提供支持
- 研究三个可以简化客户端存储的库：Lockr、Dexie和localForage
- 使用多种存储技术构建一个简单可用的应用程序

Raymond Camden是IBM的一名Developer Advocate，关注MobileFirst平台、混合移动开发、Node.js、HTML5和Web标准。他也是一名作家，并在许多会议和用户组中做过各种主题的演讲。欢迎访问他的博客：http://www.raymondcamden.com。

"客户端数据日益成为现代Web应用的一个重要组成部分，无论平台是桌面、移动Web，还是混合移动。Camden完成了一项了不起的工作，他不仅全面呈现了可供开发人员选用的技术，还提供了实例，让这个话题既有趣又实际。"

——Brian Rinaldi
Telerik开发者关系团队

WEB DEVELOPMENT/DESIGN

封面设计：Randy Comer 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／程序设计／前端开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-45014-2
定价：39.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks